

Catálogo Incompleto de Estilos Arquitectónicos

Maximiliano Cristiá
Ingeniería de Software
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario

2006

Resumen

Este documento describe los estilos arquitectónicos que se muestran en la materia Ingeniería de Software (LCC-FCEIA-UNR) en forma de catálogo. En consecuencia no es un catálogo completo ni esperamos que lo sea. Además, algunas de las características que son necesarias para describir un estilo arquitectónico tampoco se incluyen pues las consideramos innecesarias para un curso de Ingeniería de Software. Sin embargo, consideramos que cada uno de los estilos consignados está suficientemente bien descrito y documentado incluso para ser usado por profesionales y no solo por estudiantes.

Índice general

1. Organización del Catálogo	4
2. Invocación Implícita	6
2.1. Nombres	6
2.2. Propósito	6
2.3. Aplicabilidad y Ejemplos	6
2.4. Componentes	7
2.5. Conectores	7
2.6. Patrones Estructurales	8
2.7. Modelo Computacional Subyacente	8
2.8. Invariantes Esenciales	9
2.9. Metodología de Diseño	9
2.10. Análisis (Ventajas y Desventajas)	10
2.11. Documentación	10
2.12. Especializaciones Comunes	11
2.12.1. Definición de eventos	11
2.12.2. Estructura de los eventos	12
2.12.3. Asociación de eventos	12
2.12.4. Política de entrega	12
2.13. Deformaciones Comunes	13
3. Tubos y Filtros	14
3.1. Nombres	14
3.2. Propósito	14
3.3. Aplicabilidad y Ejemplos	14
3.4. Componentes	15
3.5. Conectores	15
3.6. Patrones Estructurales	15
3.7. Modelo Computacional Subyacente	15
3.8. Invariantes Esenciales	16
3.9. Metodología de Diseño	16
3.10. Análisis (Ventajas y Desventajas)	17
3.11. Documentación	19
3.12. Especializaciones Comunes	22
3.12.1. <i>Pipelines</i>	22
3.12.2. Puertos y tubos tipados	22
3.13. Deformaciones Comunes	22

3.13.1. Repositorio común para los datos	23
3.13.2. Sintonizadores	23
3.13.3. Ciclos	24
4. Sistemas Estratificados	25
4.1. Nombres	25
4.2. Propósito	25
4.3. Aplicabilidad y Ejemplos	25
4.4. Componentes	25
4.5. Conectores	26
4.6. Patrones Estructurales	26
4.7. Modelo Computacional Subyacente	27
4.8. Invariantes Esenciales	27
4.9. Metodología de Diseño	27
4.10. Análisis (Ventajas y Desventajas)	29
4.11. Documentación	29
4.12. Especializaciones Comunes	29
4.13. Deformaciones Comunes	30
5. Control de Procesos	31
5.1. Nombres	31
5.2. Propósito	31
5.3. Aplicabilidad y Ejemplos	31
5.4. Componentes	32
5.5. Conectores	32
5.6. Patrones Estructurales	32
5.7. Modelo Computacional Subyacente	32
5.8. Invariantes Esenciales	33
5.9. Metodología de Diseño	33
5.10. Análisis (Ventajas y Desventajas)	34
5.11. Documentación	35
5.12. Especializaciones Comunes	35
5.13. Deformaciones Comunes	35
6. Blackboard Systems	36
6.1. Nombres	36
6.2. Propósito	36
6.3. Aplicabilidad y Ejemplos	36
6.4. Componentes	37
6.5. Conectores	39
6.6. Patrones Estructurales	39
6.7. Modelo Computacional Subyacente	40
6.8. Invariantes Esenciales	41
6.9. Metodología de Diseño	41
6.10. Análisis (Ventajas y Desventajas)	43
6.11. Documentación	44
6.12. Especializaciones Comunes	44

6.13. Deformaciones Comunes	44
7. Cliente/Servidor de Tres Capas	45
7.1. Nombres	45
7.2. Propósito	45
7.3. Aplicabilidad y Ejemplos	45
7.4. Componentes	46
7.5. Conectores	47
7.6. Patrones Estructurales	47
7.6.1. Las capas físicas	48
7.6.2. Las capas lógicas	48
7.6.3. La forma más común de distribución	49
7.6.4. Distribución de la presentación	49
7.6.5. Distribución del procesamiento de la aplicación	50
7.6.6. Distribución de datos	53
7.7. Modelo Computacional Subyacente	55
7.8. Invariantes Esenciales	55
7.9. Metodología de Diseño	55
7.10. Análisis (Ventajas y Desventajas)	58
7.11. Documentación	58
7.12. Especializaciones Comunes	58
7.13. Deformaciones Comunes	59

Capítulo 1

Organización del Catálogo

El catálogo describe seis estilos arquitectónicos. Cada estilo se describe o documenta por medio de una serie de secciones cada una de las cuales explica una característica particular del estilo tal como se hace en [1, 2]. Las secciones fueron seleccionadas combinando ideas propias con los libros recién citados más [3] con el objetivo de brindar la descripción más completa posible de cada estilo. La información para confeccionar el catálogo se obtuvo de [2, 3, 4, 5, 6]. A continuación listamos las secciones con que se explica cada estilo.

Nombres Lista los nombres usuales del estilo.

Propósito Cinco renglones que dan una descripción general del estilo y la clase de problemas que se intenta resolver utilizándolo.

Aplicabilidad y Ejemplos Enumeración de las principales situaciones en las que resulta conveniente utilizar el estilo; posibles dominios de aplicación en los cuales el estilo preste una importante utilidad; ejemplos comunes de uso del estilo.

Componentes Los elementos computacionales activos (componentes) que forman parte del estilo (por ejemplo, filtros, fuentes de conocimiento, etc.).

Conectores Los principales conectores que se utilizan en el estilo para que los componentes puedan interactuar entre sí.

Patrones Estructurales Se listan los principales patrones estructurales que se dan en el estilo, es decir, las combinaciones de componentes y conectores que dan forma al estilo. Se indican, además, las restricciones sobre estos patrones.

Modelo Computacional Subyacente Esta sección incluye una explicación del funcionamiento en tiempo de ejecución de un sistema que implementa el estilo. En algún sentido es la sección que explica la semántica del estilo.

Invariantes Esenciales En la mayoría de los estilos arquitectónicos los distintos componentes y conectores deben interactuar de forma que se verifiquen ciertas propiedades; en muchos casos si una de tales propiedades no se verifica entonces puede ocurrir que el sistema no esté implementando un diseño en el estilo en cuestión. En esta sección se documentan esas propiedades que se denominan invariantes del estilo pues se espera que todos los diseños las verifiquen.

Metodología de Diseño En algunos casos, al decidir utilizar un determinado estilo arquitectónico, es conveniente generar el diseño aplicando una metodología particular. Por lo tanto, es interesante explicar esta metodología.

Análisis (Ventajas y Desventajas) Aplicar un estilo arquitectónico supone aprovechar sus ventajas pero también tener en cuenta sus desventajas. Sin dudas, todos los estilos son buenas formas de descomponer estructuralmente un sistema pero ninguno de ellos carece de problemas.

Documentación En esta sección mostraremos las vistas más comunes a la hora de documentar el uso de un estilo en un sistema en particular; además, extenderemos la notación 2MIL para que tenga en cuenta las características particulares de cada estilo.

Especializaciones Comunes En todas las secciones anteriores describiremos cada una en su forma más general posible, en esta sección mostraremos cuáles son las especializaciones que se aplican con mayor frecuencia. Una especialización de un estilo, es un estilo en sí mismo al cual se le han agregado más invariantes, es decir más restricciones.

Deformaciones Comunes Es muy raro que al diseñar un sistema real se aplique un estilo tal cual es, lo más común es que sea necesario relajar alguna de sus restricciones o invariantes. Al hacerlo se está deformando el estilo. En esta sección comentaremos las deformaciones más comunes de cada estilo.

Capítulo 2

Invocación Implícita

2.1. Nombres

Tool Abstraction - Eventos - Publicar y Suscribirse

2.2. 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.

El estilo de Invocación Implícita (II) elimina estas dos 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.

2.3. Aplicabilidad y Ejemplos

Se sugiere aplicar este estilo cuando:

- 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.
- Los componentes no requieran pasarse grandes cantidades de información entre sí.
- No se está seguro que las interfaces de los componentes sean las que actualmente están definidas y se espere se requieran cambios en ellas.
- Se quiera mantener muy independientes a los distintos sub-sistemas de un sistema; posiblemente los sub-sistemas implementen otros estilos.

Algunos ejemplos donde se ha aplicado con éxito este estilo son:

- Los entornos de desarrollo Field [7] y Dessert [8]
- Forest [9]
- HP Softbench [10]
- Sistemas reactivos
- Interfaces gráficas de usuario (GUI)
- Entornos integrados de desarrollo (IDE); integración de aplicaciones.
- Planillas de cálculo
- *Triggers* en bases de datos
- *Blackboard Systems*

2.4. 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 del sistema.
4. Las *toolies* implementan FSM de muy pocos estados.
5. El tiempo de vida de una *toolie* suele ser menor al de un TAD.

Por otro lado, la infraestructura de un sistema de II 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 descritos. Es muy probable que este componente venga dado por el entorno donde ejecutará o se desarrollará el sistema.

2.5. Conectores

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.

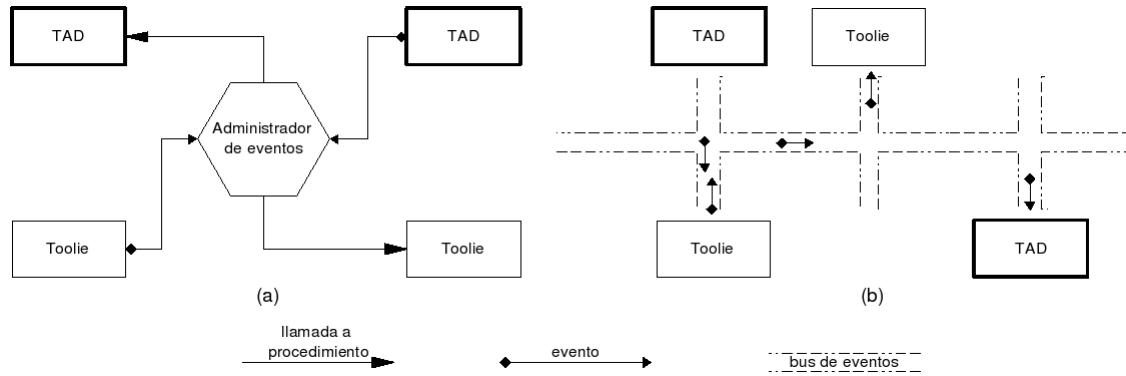


Figura 2.1: El gráfico (a) corresponde al caso en que hay un administrador de eventos, en tanto que el (b) corresponde a un *bus* de eventos.

- 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.

2.6. 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 2.1.

2.7. 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.

2.8. 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.

2.9. 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. Claramente, cada TAD oculta una estructura de datos importante del sistema. Dentro de esta fase podemos distinguir dos etapas:
 - a) Definir las subrutinas del TAD. Esto se lleva a cabo siguiendo la metodología del DBOI, DTAD o DOO según se explicó en clase.
 - b) 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. Buenos candidatos en muchos sistemas son: anunciar el inicio de la ejecución de cada subrutina, anunciar el fin de la ejecución de cada subrutina, anunciar la llegada de datos especiales (por ejemplo datos de control), anunciar estados especiales del TAD (como por ejemplo, saturación del espacio de almacenamiento, que una variable de estado llegó a algún valor de interés, etc.).
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. Por lo general esperan unos pocos eventos hasta que cierta condición se cumple, luego de lo cual anuncian uno o más eventos usualmente destinados a que se invoque algún servicio de algún TAD.

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. Es en este paso cuando los diversos componentes del sistema están dispuestos a cooperar entre sí.

En el caso en que la comunicación entre los componentes se conciba por medio de un bus de eventos, la configuración del sistema consiste en la descripción de las situaciones en las cuales cada componente aceptará cada uno de los eventos del sistema.

2.10. Análisis (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 misma naturaleza el estilo de II:

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

2.11. 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 administrador 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.

2MIL. 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.

2MIL. 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 (*evento*, *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:

callonevent *evento* p_1, \dots, p_n **calls** *subr*(p_{i_1}, \dots, p_{i_k})

donde p_1, \dots, p_n son los posibles parámetros del evento *evento* y p_{i_1}, \dots, p_{i_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 (los nombres no tienen que coincidir necesariamente con los utilizados en la correspondiente cláusula **announces**¹. Más suscripciones se pueden agregar en lka 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.

2.12. Especializaciones Comunes

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

2.12.1. Definición de eventos

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

Vocabulario fijo de eventos. No es posible definir nuevos eventos. El sistema provee un cierto conjunto de eventos y solo se pueden usar esos.

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.

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.

Declaración centralizada de eventos. Los eventos se declaran y se declaran todos en el mismo módulo del sistema.

Declaración distribuida de eventos. Los eventos se declarar pero cada evento se lo puede declarar en un módulo diferente al resto.

¹Notar que de esta forma un editor 2MIL puede chequear que haya consistencia entre las cláusulas **announces** y **callonevent** de los distintos módulos.

2.12.2. Estructura de los eventos

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

Nombres simples. Los eventos no pueden tener parámetros.

Lista de parámetros fija. Todos los eventos tienen la misma cantidad y tipo de parámetros.

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.

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.

2.12.3. 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:

Asociación estática. Los suscriptores se establecen en tiempo de compilación. Es decir el programador establece la configuración del sistema en el momento de la compilación.

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:

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.

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.

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.

2.12.4. 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:

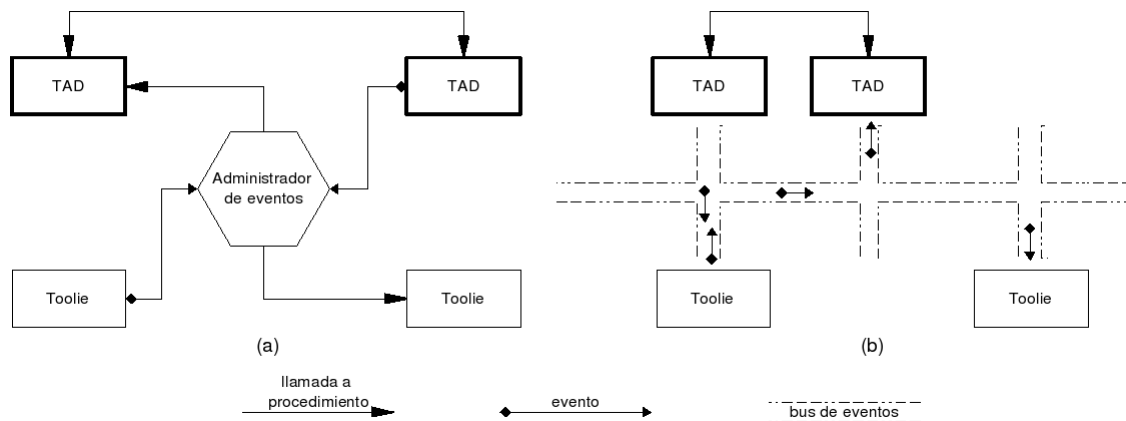


Figura 2.2: Es muy común que los TADs se comuniquen entre sí, también, vía llamada a procedimiento para comunicarse datos y ya que sus interfaces son las más estables.

Entrega completa. Un evento es anunciado a todos sus suscriptores.

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 (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).

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.

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.

2.13. 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. Esto se resume en la Figura 2.2.

Capítulo 3

Tubos y Filtros

3.1. Nombres

No hay otros nombres conocidos.

3.2. 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 recomblando, eliminando y agregando filtros.

3.3. Aplicabilidad y Ejemplos

Se sugiere aplicar este estilo en los siguiente casos:

- Procesamiento de señales
- Procesamiento de imágenes o sonido
- Compiladores
- Procesamiento de cadenas
- 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*).
- 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.

Ejemplos de aplicación conocidos:

UNIX. Se lo utiliza comúnmente para la programación de *shell scripts*.

CMS Pipelines. Es una extensión del sistema operativo CMS de IBM (usando en *mainframes*) que soporta tubos y filtros. En lugar de transmitir caracteres se transmiten registros.

LASSPTools. Es un conjunto de herramientas para análisis numérico y graficación. Cada una es un filtro que puede ser conectado con otro usando los *pipe* de UNIX.

TextPipe. Según el fabricante: "TextPipe makes it fast and easy to convert, transform and re-purpose data in text files". Ver <http://www.crystalsoftware.com.au/textpipe.html>.

3.4. 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. 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.

3.5. 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).

3.6. Patrones Estructurales

Un sistema de tubos y filtros forma un grafo dirigido acíclico desde la *fuentes* 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.

3.7. 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 consume 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.

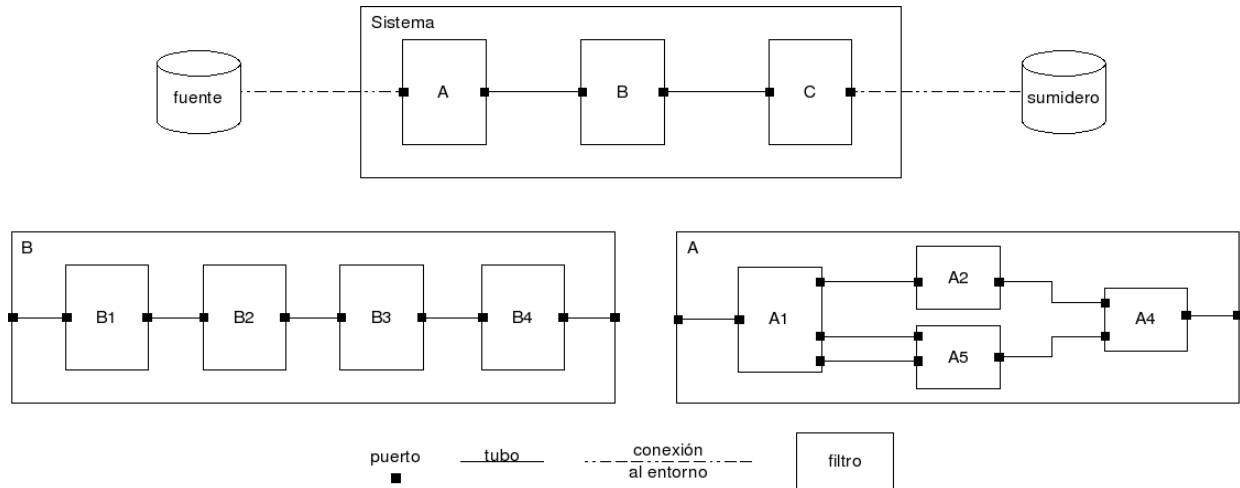


Figura 3.1: El sistema consta de tres filtros, dos de los cuales se componen de otros filtros.

3.8. 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.

3.9. Metodología de Diseño

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

1. Dividir la tarea del sistema en una secuencia de etapas de procesamiento como se muestra en la Figura 3.1. Cada etapa debe depender únicamente de la entrada que recibe. Apuntar a obtener una división jerárquica (composicional) en el sentido mostrado en la figura mencionada (ver además sección 3.10).
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. Por ejemplo, si la representación común es CHAR pero por un tubo viajan números enteros (es decir el flujo de caracteres se compone exclusivamente de dígitos), entonces habrá filtros que deberán convertir secuencias de CHAR a INT para, por ejemplo, poder operar aritméticamente, y luego volver a convertirlos en CHAR para emitir por los puertos de salida.

Un punto menor pero crucial es definir cómo se indicará el fin de datos.

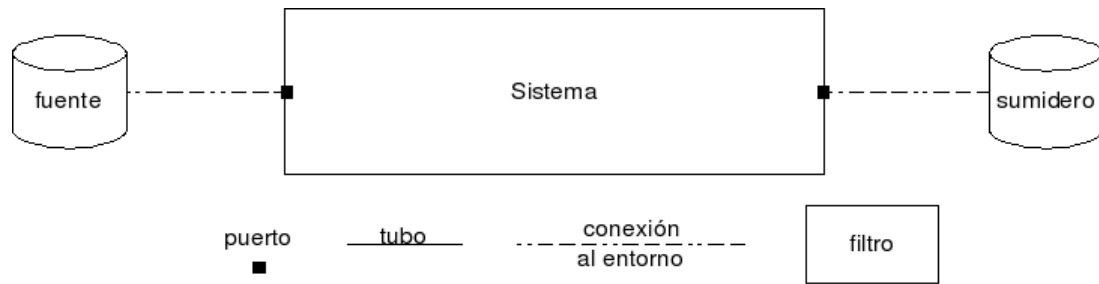


Figura 3.2: Un sistema de tubos y filtros es en sí mismo un filtro.

- 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.

- 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, ver sección 3.12) y se da una especificación funcional de la máquina de estados que implementará.

Si los filtros se implementan como procesos separados, tener en cuenta que el cambio de contexto entre filtros y la copia de datos entre espacios de direcciones puede tener un impacto severo en el desempeño.

Si los filtros poseen sintonizadores (ver sección 3.13), entonces se deben documentar como una tercera interfaz compuesta por procedimientos (ver también sección 3.11).

- Diseñar el manejo de errores. Dado que un sistema de TF 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. Por ejemplo, UNIX define un puerto específico (presente en todos los filtros) para comunicar errores.

3.10. Análisis (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. Por ejemplo el sistema de la Figura 3.1 es el filtro de la Figura 3.2

En términos más formales, si pensamos a un filtro como una función de sus puertos de entrada en sus puertos de salida tendríamos que, por ejemplo:

$$\begin{aligned}
 F &: \Gamma^n \rightarrow \Gamma^m \\
 G &: \Gamma^v \rightarrow \Gamma^w
 \end{aligned}$$

son dos filtros el primero de los cuales tiene n puertos de entrada y m de salida, el segundo v de entrada y w de salida y todos los puertos trabajan con datos de "tipo" Γ (Γ^n es el producto cartesiano de n veces Γ)¹. Entonces si los primeros k puertos de salida del filtro F se conectan con los primeros k puertos de entrada del filtro G (con $k \leq \min(m, v)$)² tenemos un nuevo filtro H definido por la composición matemática de F con G :

$$H = G \circ F : \begin{array}{ccc} \Gamma^{n+v-k} & \rightarrow & \Gamma^{w+m-k} \\ (x_1, \dots, x_{n+v-k}) & \rightarrow & H(x_1, \dots, x_{n+v-k}) \end{array}$$

donde

$$H(x_1, \dots, x_{n+v-k}) = (F(x_1, \dots, x_n).k + 1, \dots, F(x_1, \dots, x_n).m, G(F(x_1, \dots, x_n).1, \dots, F(x_1, \dots, x_n).k, x_{k+1}, \dots, x_{n+v-k}))$$

donde la notación $F(x_1, \dots, x_n).k$ representa la k -ésima componente del vector $F(x_1, \dots, x_n)$.

La posibilidad de contar con una noción de composición tan formal, clara y poderosa no es un detalle menor a la hora de construir sistemas. Son muy pocos los ámbitos en los cuales se dispone de tal herramienta. Esto se relaciona fuertemente con el paso 1 de la sección 3.9.

- 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 paliativas.
- 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.

¹Obviamente las n instancias de Γ deberán ser reemplazadas por Γ_i ($i : 1..n$) en el caso de trabajar con puertos y tubos tipados, ver sección 3.12.

²La restricción que sean los primeros puertos puede eliminarse pero torna la formalización más complicada.

- 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 (ver sección 3.13).
- 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 redundando en una pérdida de desempeño. Para mitigarlo se puede recurrir a puertos y tubos tipados, ver sección 3.12.
- 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 en abrazo mortal (*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 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.

3.11. Documentación

La documentación específica para diseños basados en TF 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 como el que se muestra en la Figura 3.4. 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 como se muestra en la Figura 3.3 (para comprenderla leer toda la sección).

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).

Module	Receptor
exportsproc	tiempo(i Int)
inports	datos,paquete:Char
outports	regla,contenido:Char ip:Int

Module	FiltrarRegla
inports	regla:Char
outports	regla:Char

Module	Combinador
exportsproc	establecerCombinacion(i TipoCombinacion)
inports	ip:Int contenido:Char
outports	combinacion:Int

Module	TuboInt is Tubo(Int)
---------------	-----------------------------

Module	TuboChar is Tubo(Char)
---------------	-------------------------------

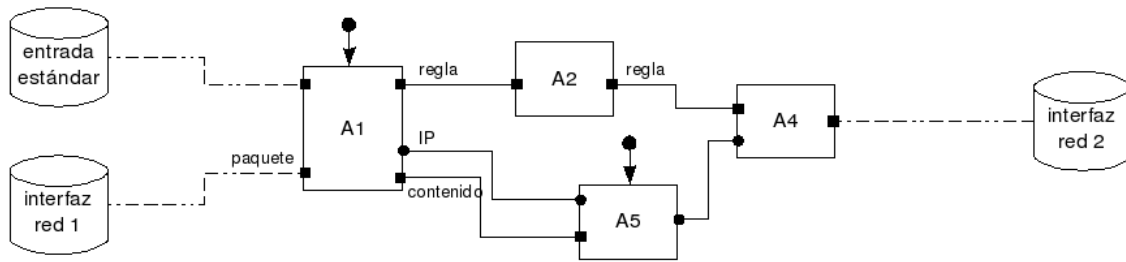
Declaraciones

A1:Receptor;
A2:FiltrarRegla;
A5:Combinador;
t1,t2:TuboChar;
t3:TuboInt;

Asociaciones

A1.regla **as** t1.write;
A1.ip **as** t3.write;
A1.contenido **as** t2.write;
A2.regla **as** t1.read;
A5.ip **as** t3.read;
A5.contenido **as** t2.read;

Figura 3.3: Representación textual de una parte del sistema de TF de la Figura 3.4.



Referencias

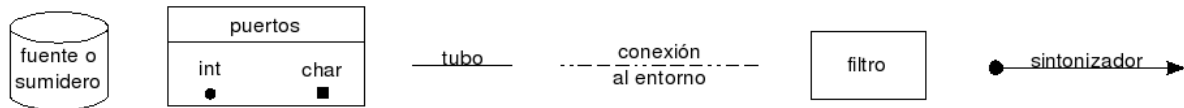


Figura 3.4: Ejemplo de diagrama canónico para TF. Es esencial incluir la referencia para cada símbolo.

- 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 (ver sección 3.13) 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.
- Si se usan puertos y tubos tipados (ver sección 3.12) se anota el tipo sobre el tubo o sobre la conexión al medio ambiente. Alternativamente, pueden utilizarse símbolos diferentes como en la Figura 3.4.

Especialización/Deformación seleccionada. Documento que describe las cuestiones de diseño que quedan abiertas en la descripción del estilo. Por ejemplo: uso de sintonizadores (ver sección 3.12), comportamiento de bloqueo en tubos, implementación como procesos o hilos de ejecución, manejo de errores, etc.

2MIL. Se agregan las cláusulas **inports** y **outports**:

inports nombre[:Tipo]

outports nombre[:Tipo]

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

2MIL. 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.

3.12. Especializaciones Comunes

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

3.12.1. *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.

3.12.2. 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).

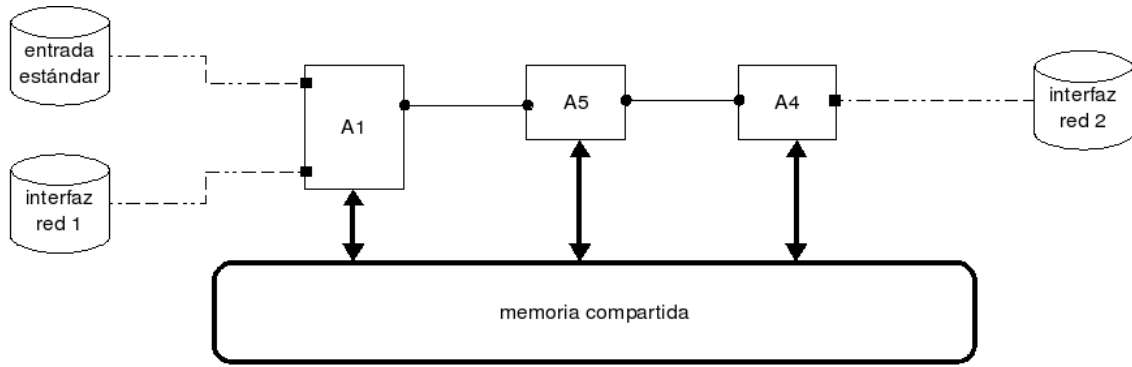
Si bien esta especialización puede dar lugar a menor flexibilidad (ya que ahora los filtros dependen más del contexto que antes) permite mejorar la eficiencia en la transmisión de información pues no requiere que los filtros hagan una y otra vez conversiones de datos. Por ejemplo, si por un tubo viajan únicamente caracteres que son dígitos entonces el tubo se define de tipo entero y por lo tanto los filtros a sus extremos emiten y reciben enteros sin necesidad de hacer conversiones.

Además, reduce la posibilidad de errores de programación al habilitar un uso más extendido de tipos. El compilador podrá detectar que por un puerto de tipo T_1 se está intentando enviar un dato de tipo T_2 .

Finalmente, en el mismo sentido, permite transmitir tipos complejos (por ejemplo, Imagen, Sonido, Matriz, etc.) que pueden implementarse como TADs u objetos lo que agrega aun más orden al aplicar el principio de ocultación de información.

3.13. Deformaciones Comunes

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



Referencias

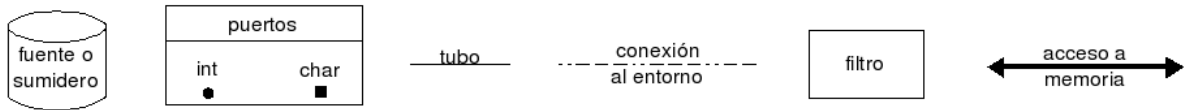


Figura 3.5: Un repositorio común donde los filtros escriben y leen datos.

3.13.1. 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 prohibitiva, se sugiere utilizar un repositorio común de datos, ver Figura 3.5.

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 (en la figura mencionada no se representa; podría ser un semáforo que proveyera exclusión mutua el cual, a su vez, es provisto por el sistema operativo). Aun así ningún filtro debe conocer la identidad de los otros.

Al aparecer el repositorio común aparece un nuevo conector que debe ser documentado y explicado. Por ejemplo, si se usa un semáforo debe explicarse cómo utilizarlo, quién lo provee, dónde se declara, etc.

El repositorio no necesariamente debe reemplazar a todos los tubos, ambos mecanismos de comunicación pueden utilizarse según convenga como se muestra en la Figura 3.5 en el cual los enteros se transmiten a través de tubos pero los caracteres se acceden y modifican en un sector de memoria compartida.

3.13.2. 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.

3.13.3. 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 fundamento 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 [2].

Capítulo 4

Sistemas Estratificados

4.1. Nombres

Máquinas abstractas jerárquicas.

4.2. 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.

4.3. Aplicabilidad y Ejemplos

- Máquinas Virtuales (Java Virtual Machine, por ejemplo)
- API's (*Application Programming Interface*).
- 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).

4.4. 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.



Figura 4.1: Dos representaciones gráficas para la topología de los sistemas estratificados.

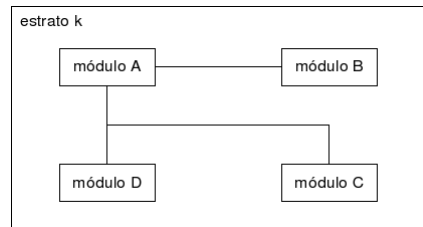


Figura 4.2: Los estratos pueden estar estructurados en módulos.

4.5. 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.

4.6. 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 $k - 1$. Gráficamente existen varias opciones como se muestra en la Figura 4.1, además puede haber variantes del estilo que se describen como en la Figura 4.3. En la Figura 4.2 puede observarse cómo un estrato puede estar compuesto por varios módulos.

En cualquier caso 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*.

4.7. 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 estrato 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.

4.8. 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.

4.9. Metodología de Diseño

Los pasos que se describen a continuación describen una de las formas de diseñar sistemas estratificados. Puede ocurrir que no todos los pasos sean necesarios o que no se sigan en el orden aquí indicado.

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 contruidos 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 forzosamente) una relación de abstracción. Si el esquema abstracción se fuerza (o bien porque no existe o porque no se lo supo encontrar) los primeros cambios no necesariamente encajarán en él y por lo tanto se tendrá que violar el invariante esencial del estilo.
6. Especificar la interfaz de cada estrato. Hay tres estrategias.

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 [1].

Caja gris. Significa que k sabe de la existencia de los módulos que componen a $k - 1$.

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 [1].
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 (ver punto 10). Entonces, para tener comunicación hacia arriba sin acoplamiento, se pueden usar *callbacks*, eventos o el patrón de diseño Command [1].
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.

La comunicación de errores que se produce espontáneamente y no como consecuencia de una llamada se puede implementar con *callbacks*, eventos o el patrón de diseño Command.

4.10. Análisis (Ventajas y Desventajas)

Entre las ventajas tenemos las siguientes.

- Reuso de los estratos.
- Permite la estandarización (POSIX por ejemplo).
- 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.

4.11. 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. Las figuras 4.1 y 4.3 muestran algunas alternativas.

2MIL. 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.

2MIL. 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.

4.12. Especializaciones Comunes

No se han documentado.

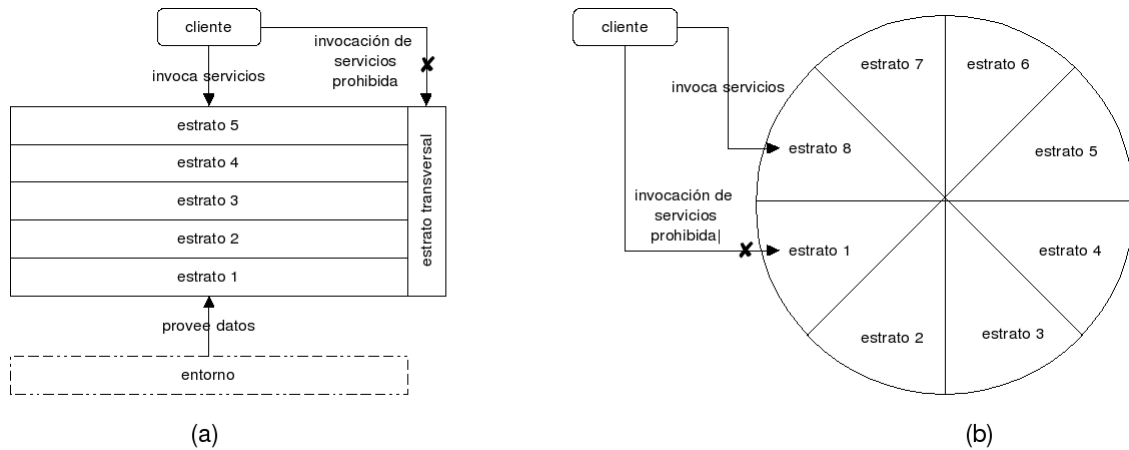


Figura 4.3: Representaciones gráficas de algunas variantes del estilo. En (b) cada estrato puede invocar servicios de los dos estratos adyacentes.

4.13. 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.

Algunas de estas variantes se grafican en la Figura 4.3.

Capítulo 5

Control de Procesos

5.1. Nombres

No hay otros nombres conocidos.

5.2. 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.

5.3. Aplicabilidad y Ejemplos

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)

Los ejemplos conocidos hasta el momento son:

- Automatización de procesos industriales.
- Sistemas SCADA.
- Aplicaciones de control industrial en general.

La mayoría de estos sistemas se implementan como sistemas *embebidos* o *embarcados*.

5.4. 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.

5.5. Conectores

No están claramente documentados en la literatura. Utilizaremos los siguientes:

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**.

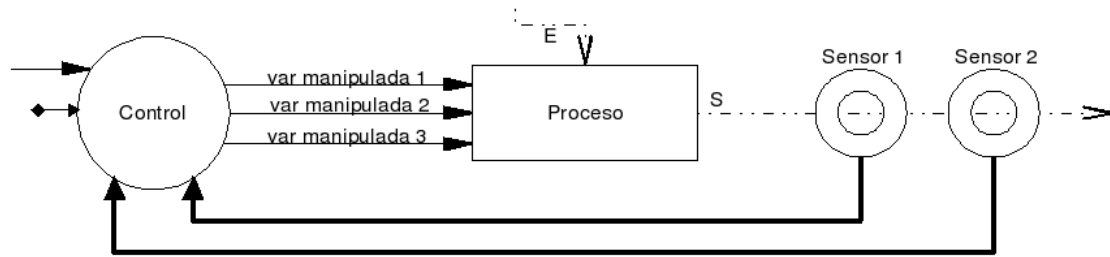
Tubo. Se utilizarán para comunicar los datos entre los **Sensores** y el **Control**.

5.6. Patrones Estructurales

Existe un único patrón estructural que se muestra en la Figura 5.1. 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.

5.7. 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



Referencias

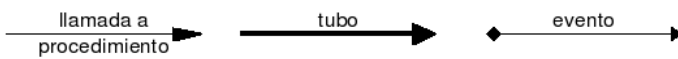


Figura 5.1: Diagrama canónico para CP. Las líneas de puntos representan la entrada y la salida del proceso físico, y no del componente **Proceso**.

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**.

5.8. 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.

5.9. 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 (aquí tener en cuenta las nociones básicas de "tipo" e "instancia") o según sus diferentes funciones.

En el mismo sentido un algoritmo de control complejo puede ser dividido en partes o los datos que utiliza pueden encapsularse en objetos o TADs, etc. 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 sub-módulos 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.

5.10. Análisis (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.

5.11. Documentación

Diagrama canónico. Se documenta gráficamente como se muestra en la Figura 5.1.

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.

5.12. Especializaciones Comunes

No se conocen.

5.13. Deformaciones Comunes

No se conocen.

Capítulo 6

Blackboard Systems

6.1. Nombres

No hay otros nombres conocidos.

6.2. 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.

6.3. Aplicabilidad y Ejemplos

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.).
- Vigilancia.
- 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.).

Algunos sistemas donde se ha aplicado este estilo son:

- The Hearsay-II Speech-Understanding System.

- GBBopen Project (<http://www.gbbopen.org/>).
- El componente PLAN del Sistema de Control de Misión para el RADARSAT-1, un satélite de observación terrestre desarrollado por Canadá para monitorear los cambios medioambientales y los recursos naturales del planeta.
- Copycat, un modelo para hacer analogías basado en el concepto de *parallel terraced scan*, desarrollado por Douglas Hofstadter, Melanie Mitchell, y otros en el Center for Research on Concepts and Cognition, Indiana University Bloomington (http://en.wikipedia.org/wiki/Copycat_%28software%29).
- Proyecto Numbo.
- Psyclone AIOS, una plataforma para desarrollar sistemas autónomos y de automatización complejos (<http://www.cmlabs.com/psyclone/>).
- GigaSpaces, implementación de un *blackboard* basada en JavaSpaces.
- PASTA es un proyecto financiado por DARPA para investigar sistemas de potencia eficientes para aplicaciones de *unattended ground sensor* (UGS) (<http://pasta.east.isi.edu/>).

6.4. Componentes

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

Blackboard. Es un repositorio de datos dividido en niveles¹. 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 Principio de Ocultación de la Información 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 (usualmente las relaciones son *PARTE_DE* o *CORROBORA_A*). Sin embargo ningún nivel accede a los datos de otro nivel.

¹Estos niveles no tienen nada que ver con los niveles o estratos de los Sistemas Estratificados.

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 "especialista." "experto" en resolver un aspecto del problema global. Ninguna **FC** puede resolver el problema por sí sola.

La regla o algoritmo de una **FC** está condicionada por una precondición. Todas las **FC** tienen interfaces muy semejantes (en general son idénticas pero pueden diferenciarse en la forma que son parametrizadas o inicializadas). 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.

Como se explica en la sección 6.6 las **FC** consultan y modifican el **Blackboard** por lo que deben conocer su estructura, semántica e interfaz.

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 de razonamiento hacia adelante 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 de razonamiento hacia atrás implica que la **FC** consulta ciertos niveles del **Blackboard** y hace su aporte escribiendo en un nivel inferior a todos ellos.

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.

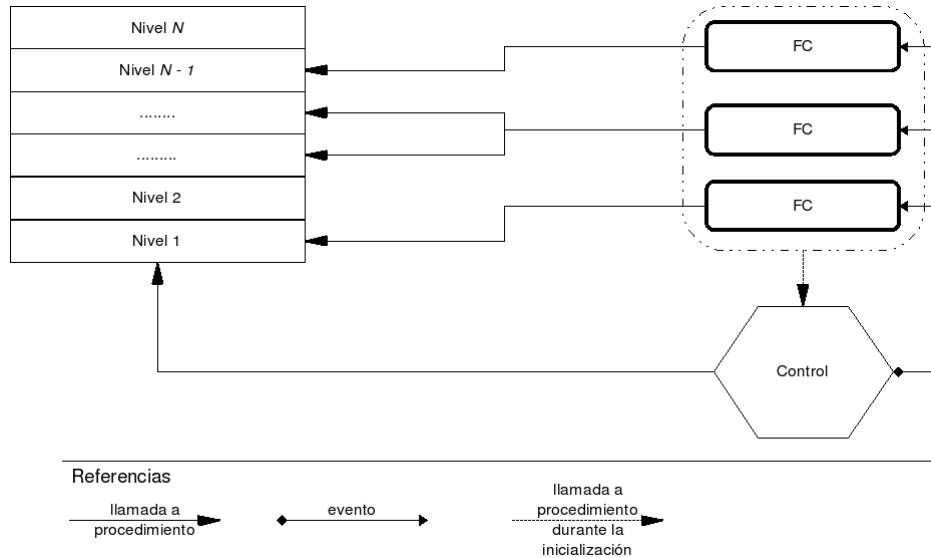


Figura 6.1: Los sistemas BS se construyen con una única instancia de un **Blackboard**, una o más **FC** y una única instancia de un subsistema de **Control**.

6.5. Conectores

El estilo BS no define nuevos tipos de conectores sino que utiliza eventos y llamada a procedimiento como se muestra en la Figura 6.1. Los eventos, en realidad, se implementan con una forma de invocación implícita ya que lo que se espera que el **Control** no mantenga referencias estáticas a las **FC**.

6.6. Patrones Estructurales

El único patrón estructural admitido por el estilo BS se grafica en la Figura 6.1. 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** (sin embargo, ver la sección 6.13).

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 6.1 esta interacción se grafica como un evento dado que el **Control** no tiene referencias estáticas a las interfaces de las **FC**.

6.7. Modelo Computacional Subyacente

Iniciaremos la explicación del modelo computacional subyacente de un sistema basado en BS mediante una analogía. Supongamos que se deben detectar todos los koalas presentes en una gigantografía de un bosque australiano. La foto se ubica sobre el pizarrón de un aula donde hay expertos en el dominio del problema y una persona que actúa como coordinador. Entre los expertos hay personas que son capaces de marcar dónde hay una curva; una recta; un color específico; la línea del horizonte; un árbol u hojas de árboles a partir de curvas, rectas y colores; partes anatómicas de un koala (ojos, orejas, brazos, patas, torso, cabeza, etc.) a partir de curvas, rectas, colores, árboles, suelo y otras partes de un koala (por ejemplo puede ser que los koalas raramente estén en el suelo por lo que los expertos en encontrar partes de koalas se fijarán dónde están los árboles); koalas a partir de partes de koala, etc.

El coordinador pregunta a los expertos quiénes están en condiciones de aportar algo para solucionar el problema. Supongamos que varios de ellos levantan la mano. El coordinador analiza qué puede aportar cada uno de ellos y selecciona uno. El experto se aproxima al pizarrón y marca sobre la fotografía su aporte, luego vuelve a su asiento. El coordinador vuelve a preguntar quiénes pueden aportar algo pero ahora, posiblemente, otros expertos levanten su mano dado que hay más información sobre la fotografía. El coordinador vuelve a analizar qué puede aportar cada uno de los interesados y selecciona uno de ellos (posiblemente el mismo que antes). El experto hace su aporte y vuelve a su asiento. El coordinador repite este procedimiento hasta que no hay más progreso o no hay expertos que levanten su mano o considera que se ha resuelto el problema o no tiene más tiempo para continuar.

Habiendo leído la sección 6.6, se puede proyectar la analogía anterior sobre los componentes de un sistema basado en BS, excepto para los niveles del **Blackboard**. Las **FC** son los expertos, el **Control** es el coordinador y el **Blackboard** la gigantografía sobre el pizarrón (de aquí el nombre del estilo). Los niveles del **Blackboard** son un recurso computacional para estructurar la búsqueda de la solución final (una suerte de estrategia "dividir y conquistar") y para dar la posibilidad de llegar a soluciones parciales.

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 6.6.
2. El **Control** determina el foco de atención, consultando el **Blackboard** y las **FC** registradas.
3. El **Control** lista las **FC** cuya precondition 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 un paso a la vez, desalentando, en consecuencia, la posibilidad de un modelo computacional concurrente o paralelo (en contradicción con lo expresado en [2, página 73]). 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 mencionamos 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.

6.8. 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**.

6.9. Metodología de Diseño

La metodología de diseño al utilizar BS se basa en resolver un problema en un dominio de aplicación inmaduro 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:
 - a) Especificar qué significa precisamente una solución final.
 - b) Listar los distintos niveles de solución para el problema.
 - c) 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).
 - d) Buscar subdivisiones de las soluciones completas que pueden ser resueltas independientemente (por ejemplo encontrar un koala en una región de la foto es independiente de encontrar otro en otra región).
 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.
 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 [1] 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**. Por ejemplo, suelen aplicarse una o más heurísticas (priorizar **FC**, preferir razonamiento hacia adelante o hacia atrás, etc.) que deberían estar codificadas por separado.

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.

Dada la naturaleza experimental de este tipo de sistemas es muy probable que se deba iterar algunas veces entre los pasos 5 y 7.

6.10. Análisis (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 pero 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

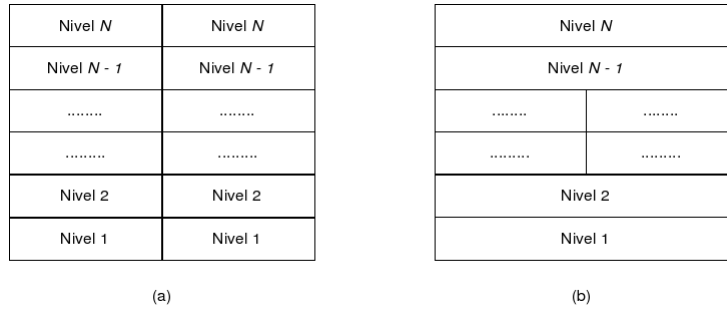


Figura 6.2: En (a) hay dos paneles disjuntos en tanto que en (b) solo algunos niveles se dividen en dos paneles.

6.11. Documentación

Diagrama canónico. Se debe documentar el diagrama canónico como se muestra en la Figura 6.1. 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. En la Guía de Módulos se debe describir la función de cada **FC**, su aporte, y todos los meta-datos 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. La Estructura de Módulos se divide naturalmente en tres módulos lógicos: **Blackboard**, **Fuentes de Conocimiento** y **Control**. A su vez el módulo **Blackboard** se subdivide en un módulo por cada nivel; en tanto que **Fuentes de Conocimiento** 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.

6.12. Especializaciones Comunes

No se han documentado especializaciones de este estilo.

6.13. 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, como se muestra en la Figura 6.2.

Capítulo 7

Cliente/Servidor de Tres Capas

7.1. Nombres

Cliente/Servidor de n capas - Arquitectura en capas - Sistemas distribuidos (aunque es un nombre demasiado genérico)

7.2. 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.

7.3. Aplicabilidad y Ejemplos

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).

Muchos de los ERP que se utilizan hoy día funcionan con una arquitectura que trata de acercarse lo más posible a este estilo. En general, debido a la existencia de grandes sistemas y datos cuya documentación es inexistente y que fueron desarrollados hace años por personal que ya no pertenece a la compañía, ninguno de ellos logra la integridad conceptual que el estilo sugiere. Por el contrario, los nuevos sistemas basados en Internet suelen tener una buena arquitectura.

Las siguientes compañías desarrollan y venden sistemas ERP que deberían seguir este estilo: SAP, Oracle Applications, Microsoft Dynamics (antes Microsoft Business Division), The Sage Group, Lawson Software, Visma, Industrial and Financial Systems. Dentro del software libre tenemos: Adempiere, Compiere, GNU Enterprise, SQL Ledger. Como la funcionalidad de un ERP no está completamente definida puede que los productos de estos vendedores varíen considerablemente.

Sitios Web como Amazon.com, EBay.com, etc. son buenos ejemplos de sistemas basados (o que deberían estar basados) en este estilo.

7.4. 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 módulo¹. Usualmente se utilizan tres capas.

Las capas no deben confundirse con los estratos de los Sistemas Estratificados [4]. 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. Los SE no suelen ser distribuidos en tanto que esa es la principal característica de los sistemas basados en Cliente/Servidor de Tres Capas en los cuales, como ya dijimos en el párrafo anterior, cada capa ejecuta en una plataforma diferente.

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. Por ejemplo, en un sistema de gestión corporativa un servidor de negocio provee servicios para procesar documentos contables en tanto que un servidor de infraestructura puede ser un RDBMS o un servidor Web.

¹Si bien la distinción entre sistema, sub-sistema y módulo no es muy clara ni precisa, con estos términos se intenta reflejar la complejidad y envergadura de cada capa.

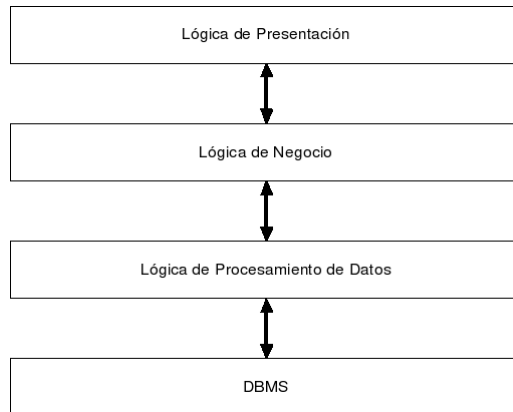


Figura 7.1: Cada rectángulo representa una capa lógica; las flechas indican comunicación bidireccional por medio de diversos conectores.

7.5. 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.

Normalmente la invocación de servicios es sincrónica: el solicitante queda bloqueado hasta que el servicio solicitado completa su tarea y retorna, posiblemente con un resultado.

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.

7.6. 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 7.1 muestra las relaciones estructurales básicas.

7.6.1. Las capas físicas

Los patrones estructurales están guiados por la topología física de la red de la corporación. Usualmente la red se compone de cientos o miles de PCs o estaciones de trabajo, decenas de servidores medianos y algunos *mainframes* o grandes servidores (a los que pueden sumarse diversos equipos de comunicación como *routers*, *firewalls*, *switches*, etc.). Las PCs se distinguen por poseer una capacidad de cómputo interesante y fundamentalmente por permitir al usuario interactuar con dispositivos de entrada/salida sofisticados. Todos estos 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. Por ejemplo, un banco de alcance nacional usualmente posee sus mainframes o grandes servidores en un único centro de cómputos² los cuales prestan servicio a las sucursales distribuidas en todo el territorio; en cada una de las sucursales hay uno o dos servidores medianos que sustentan el funcionamiento de una LAN formada por entre 10 y 100 PCs.

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.

7.6.2. Las capas lógicas

Un sistema típico en este estilo consiste de cuatro capas lógicas (representadas en la Figura 7.1):

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.; si es una compañía de seguros existirán componentes para asegurados, pólizas, siniestros, 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.

²Muchas veces contruidos para ofrecer alta redundancia.

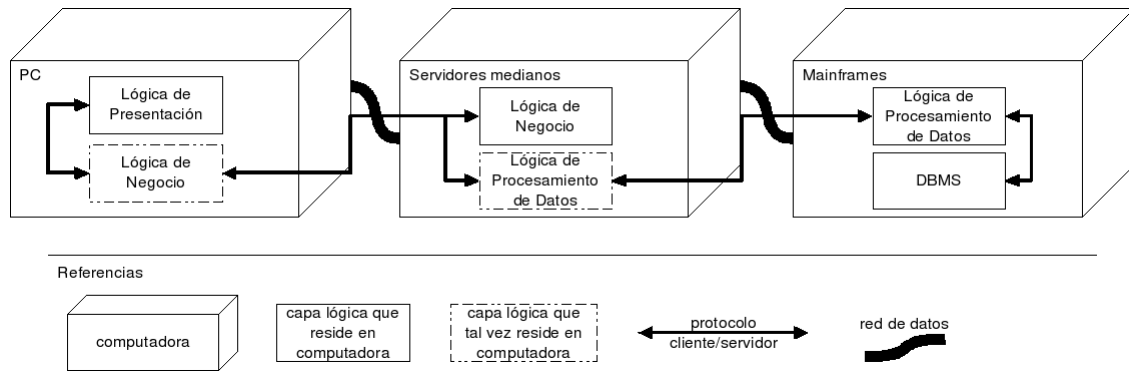


Figura 7.2: Estructura Física que representa la distribución más común de datos y aplicaciones.

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.

7.6.3. La forma más común de distribución

La forma más usual de distribuir tanto datos como aplicaciones se muestra en la Figura 7.2.

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 las tres secciones que siguen se analizarán más alternativas y se estudiarán sus ventajas y desventajas relativas.

7.6.4. Distribución de la presentación

La LP se distribuye en dos sentidos, como se muestra en la Figura 7.3:

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.

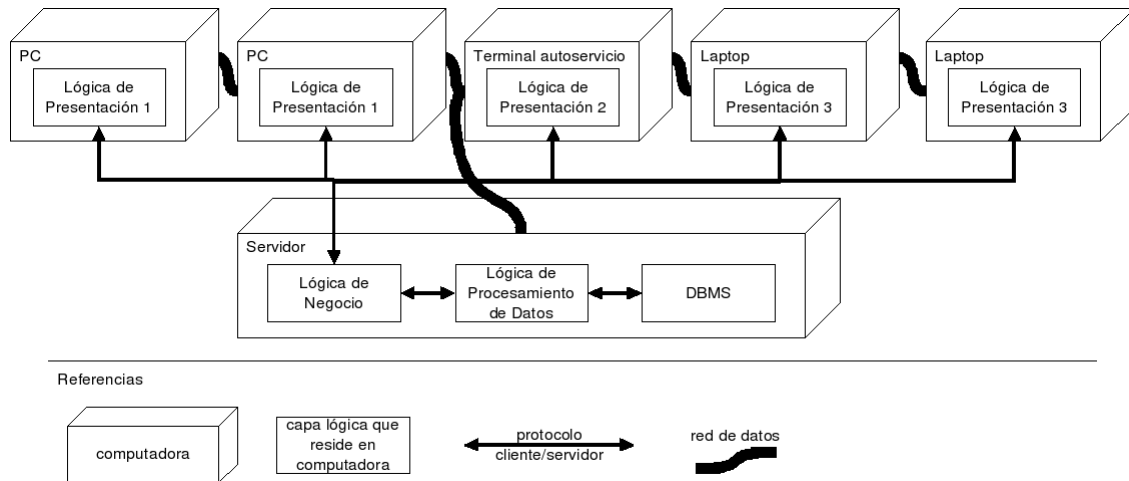


Figura 7.3: Estructura Física que representa una distribución común de las capas, donde hay varias versiones diferentes de la Lógica de Presentación.

Además, al poner la LP sobre las PCs se hace un uso mucho más eficiente del poder de cómputo de estos equipos. Antes de la aparición de este estilo arquitectónico, las PCs se utilizaban casi exclusivamente para acceder por TELNET al servidor donde ejecutaba la aplicación y se almacenaban los datos.

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 diferentes aplicaciones que requieren interacciones distintas.

7.6.5. 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 medianos.

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. Por ejemplo, dentro de esta porción entran las validaciones tempranas de la entrada y las rutinas que formatean la salida antes de pasarla a la capa de presentación. En tanto que la porción de la LN que está más vinculada con el acceso a los datos persistentes yacerá en los servidores intermedios.

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.

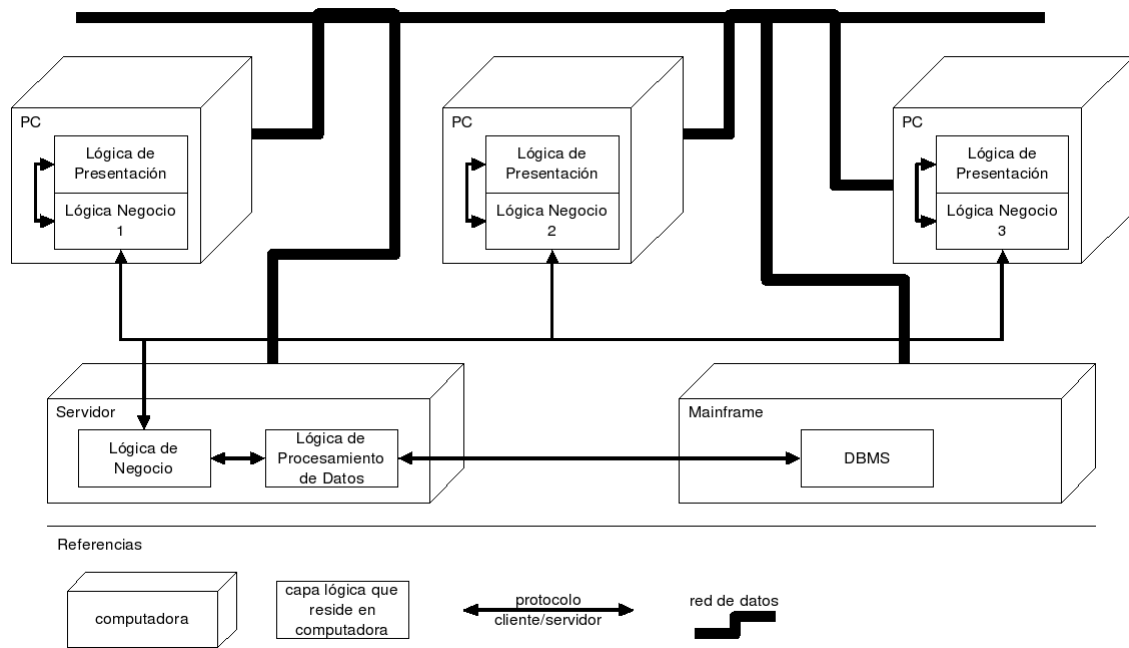


Figura 7.4: Representación de la distribución óptima del procesamiento de la aplicación.

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, como se muestra en la Figura 7.4, 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.

Una alternativa aun más compleja pero frecuentemente utilizada en grandes organizaciones se grafica de forma simplificada en la Figura 7.5. En esta arquitectura la LN está distribuida en varios servidores (más precisamente, en cada servidor ejecutan diferentes porciones de la LN) y los clientes (que pueden incluir LP y otras porciones de la LN) podrían necesitar servicios provistos por los diferentes servidores [2, página 99 y siguientes]. Si la existencia y la ubicación de cada servicio se mantiene en tablas de los clientes, un cambio en la distribución implica una actualización de todos los clientes lo que, en grandes organizaciones, no es conveniente. Por lo tanto, se incorporan servidores de infraestructura denominados *brokers* que proveen un *servicio de nombres* (NS, por su sigla en inglés). Los clientes solo conocen la existencia del servidor NS al cual se registra cada servicio provisto por la LN (indicando al menos: ubicación física (IP, por ejemplo), nombre, parámetros, mecanismo de invocación, etc.). De esta forma los clientes utilizan el *broker* para invocar servicios el cual se encarga de localizarlo, invocarlo y retornar el resultado. Por lo tanto, se desacoplan los clientes de los servidores y, fundamentalmente, se puede alterar la distribución de servicios de forma transparente y dinámica. La capa de los *brokers* puede, a su vez, distribuirse aun más [2].

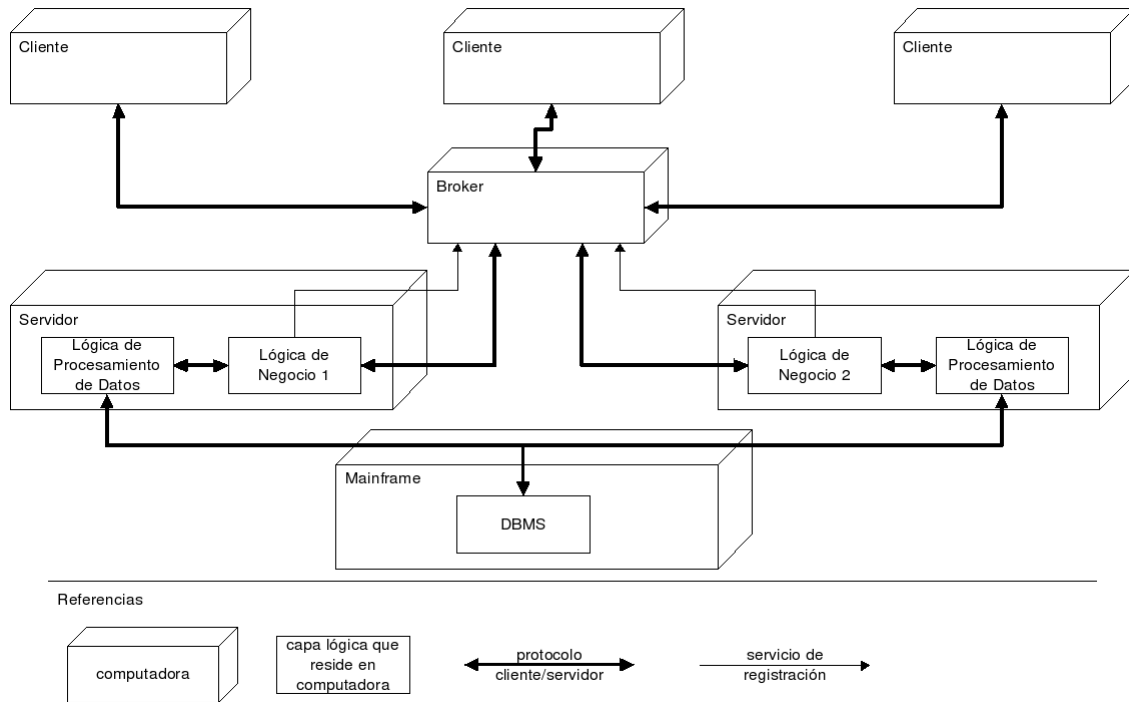


Figura 7.5: Distribución de la LN en varios servidores a los cuales, potencialmente, pueden necesitar conectarse cualquier cliente. Esta arquitectura requiere la presencia de *brokers*.

Esta alternativa del estilo CS3C, llamada *Broker* por [2], ha evolucionado a lo que se conoce actualmente como *arquitecturas orientadas a servicio* (SOA, por la sigla en inglés). En SOA los clientes y *brokers* están capacitados para utilizar nuevos servicios que antes desconocían por completo. Esto se logra a través de protocolos y estándares que permiten especificar y descubrir servicios. La tecnología dominante para implementar este tipo de arquitecturas se conoce como *Webservices*.

Un problema que permanece en cualquiera de estas alternativas es la necesidad de sincronizar clientes y servidores para completar las transacciones (que son distribuidas por la propia naturaleza del estilo). Esto suele solucionarse utilizando conectores basados en *two-phase commit protocols*.

7.6.6. Distribución de datos

Hasta aquí se mostraron los diferentes patrones estructurales que se originan al distribuir el código del sistema. Ahora 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.

Las Figuras 7.6 y 7.7 muestran las dos posibilidades más comunes. 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. Por ejemplo, en el caso del banco que mencionábamos más arriba no tendría mucho sentido que los datos de las cuentas de los clientes de la sucursal Río Gallegos deban ser transmitidos desde Buenos Aires cada vez que

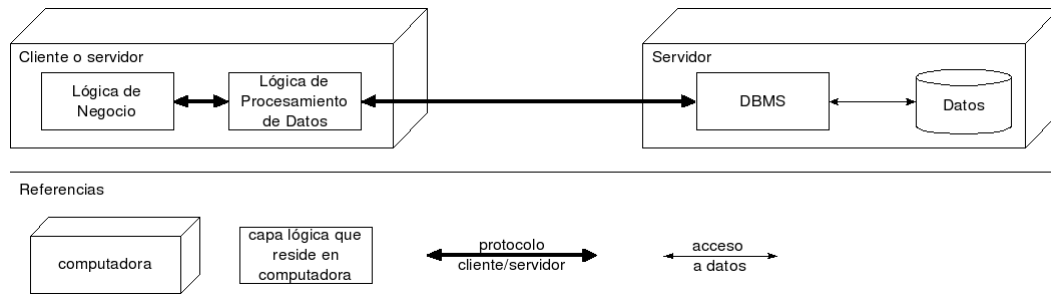


Figura 7.6: Distribución de datos llamada conexión directa.

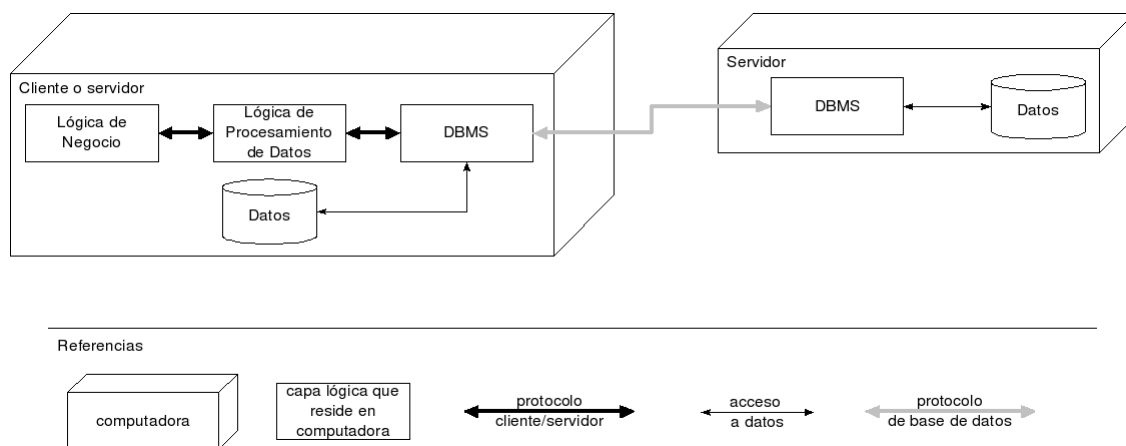


Figura 7.7: Distribución de datos llamada conexión indirecta.

uno de esos clientes solicita el saldo. Dentro de las situaciones donde este esquema es imposible tenemos el caso de vendedores que tienen computadoras portátiles y realizan ventas en la calle, actualizando los datos una vez por día o menos (claramente aquí parte de los datos corporativos yace en cada una de estas computadoras portátiles). Por otro lado, este esquema tiene una gran ventaja que es que torna mucho más simple mantener la integridad de los datos al estar todos en el mismo repositorio.

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.

7.7. 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.

7.8. 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

7.9. Metodología de Diseño

Es muy difícil hablar de una metodología de diseño para este tipo de sistemas porque hay muchos en funcionamiento y la mayoría fue desarrollado con su propia metodología. Por otro lado, siendo el estilo dominante en el dominio de aplicación al cual apuntan las más grandes empresas productoras de software existen infinidad de productos comerciales muchos de ellos con una metodología de desarrollo asociada. Además, dada la complejidad de este tipo de sistemas es muy largo explicar detalladamente una metodología.

Por lo tanto, en este catálogo 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.

Entendemos por tecnología: motores de RDBMS, versión de SQL, lenguajes de programación (seguramente habrá uno o dos correspondientes al viejo ERP y otro tanto para las nuevas porciones), sistemas operativos, protocolos de red, *brokers* (servicios de nombres, *middleware* para transacciones, etc.), sistemas de soporte (como servidores Web, LDAP, etc.), entornos de desarrollo, etc.

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. Asociado a este desconocimiento tenemos el hecho de que se suelen sub-utilizar productos complejos lo que implica que se desarrollan partes que no haría falta si se usaran los productos adquiridos en un 100%. Por ejemplo, los principales servidores de aplicaciones (WebSphere de IBM, el entorno ASP.NET de Microsoft o Tomcat) implementan módulos para control de acceso, autorización y auditoría pero muy pocos desarrolladores saben utilizarlos o saben de su existencia por lo que suelen programar rutinas de control de acceso o autenticación que están provistas por la infraestructura.

Finalmente, se debe poner especial atención a las interfaces entre estos productos. No siempre están bien documentadas, no siempre funcionan correctamente y no es simple usarlas como corresponde. Estos desacoples puede alargar considerablemente los tiempos de desarrollo, pueden requerir el rediseño de vastas porciones del sistema y pueden llevar el proyecto al fracaso.

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.
 - a) Se deben analizar todas las alternativas presentadas en la sección 7.6
 - b) Se debe seleccionar la más adecuada
 - c) Se la debe documentar con precisión
 - d) 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 como el manejo de eventos a bajo nivel, la *renderización* en pantalla, etc. Queda como tarea para el equipo de desarrollo definir el *look-and-feel* de la aplicación, los menús, formularios, ventanas 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).
5. 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.

Idealmente, luego del proceso de reingeniería debería reorganizarse el código existente para adecuarlo a un diseño más moderno que permita la incorporación de cambios sin un costo alto. Esto raramente se lleva a cabo a pesar de que en muchos casos no sería necesario reprogramar grandes porciones del sistema.

Al menos en la parte de la LN que debe ser implementada desde cero se deberían aplicar todas las consideraciones del Diseño Basado en Ocultación de Información. Esto implica, por ejemplo, aislar en módulos separados la representación y ubicación física de los datos. En otras palabras unos pocos módulos de esta capa deberían conocer el modelo de datos mantenido en el DBMS y la LPD utilizada para accederlo (como ya dijimos la LPD suele ser SQL o similar lo que en general revela más información sobre las estructuras de datos de la necesaria). Es muy importante poner empeño en esta tarea ya que uno de los cambios más frecuentes es, precisamente, cambiar el modelo de datos. Para lograr un buen diseño de lo que acabamos de mencionar y para muchas otras cuestiones se deberían aprovechar los Patrones de Diseños documentados en la literatura [1, 2].

6. 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.

7.10. Análisis (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 7.2).

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.

7.11. 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.

7.12. 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 7.6.
- 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.

³Habitualmente se utiliza la sigla COTS para designar a los componentes comprados a terceros. COTS es sigla de *commercial off-the-shelf*.

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

7.13. Deformaciones Comunes

Son innumerables y en general corresponden a errores en el diseño o a concesiones 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) [4, página 139 y siguientes]. En este estilo la asimetría entre clientes y servidores se pierde por lo que todos los componentes se convierten en *pares*⁴. 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.

⁴Aquí el término se utiliza como sinónimo de igual o colega.

Bibliografía

- [1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Patrones de diseño*. Addison Wesley, 2003.
- [2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stad, *Pattern-Oriented Software Architecture — A System of Patterns*. John Wiley Press, 1996.
- [3] M. Shaw and D. Garlan, *Software architecture: perspectives on an emerging discipline*. Upper Saddle River: Prentice Hall, 1996.
- [4] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little, *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.
- [5] D. Garlan, G. E. Kaiser, and D. Notkin, “Using tool abstraction to compose systems,” *Computer*, vol. 25, no. 6, pp. 30–38, 1992.
- [6] A. Berson, *Client/server architecture*. New York, NY, USA: McGraw-Hill, Inc., 1992.
- [7] S. P. Reiss, “Connecting tools using message passing in the Field environment,” *IEEE Softw.*, vol. 7, no. 4, pp. 57–66, 1990.
- [8] —, “The Desert environment,” *ACM Trans. Softw. Eng. Methodol.*, vol. 8, no. 4, pp. 297–342, 1999.
- [9] D. Garlan and E. Ilias, “Low-cost, adaptable tool integration policies for integrated environments,” in *SDE 4: Proceedings of the fourth ACM SIGSOFT symposium on Software development environments*. New York, NY, USA: ACM Press, 1990, pp. 1–10.
- [10] C. Gerety, “A new generation of software development tools,” Hewlett-Packard Software Engineering System Division, Tech. Rep. SESD-89-25), 1989.