Introducción a los Patrones (Diseño y Arquitectura)

Clases 12 y 13

Otoño del 2005

Prof. Sergio Ochoa

Estructura de la Presentación

- Generalidades
- Estructura de los Patrones
- Categorías de Patrones Arquitectónicos
- Patrones Arquitectónicos para Sistemas Simples
- Patrones Arquitectónicos para Sistemas Interactivos
- Patrones Arquitectónicos para Sistemas Adaptables
- Conclusiones

¿Para Qué Comunicar el Diseño?

- No reinventar soluciones a problemas conocidos.
- Reusar el conocimiento experto de diseño.
- Beneficios:
  - inexpertos pueden diseñar como expertos,
  - menos errores debido al uso de diseños ya probados,
  - los diseños probados son más fácilmente mantenibles,
  - el software puede diseñarse para tener ciertas propiedades.
- ¿Es esta comunicación exitosa?
  - sólo entre virtuosos,
  - ¿por qué?

Una historia que cambió la forma de ver el diseño….

- *The Spirit of Patterns: The Re-design of Manteo*

- Diseñador: Randolph T. Hester

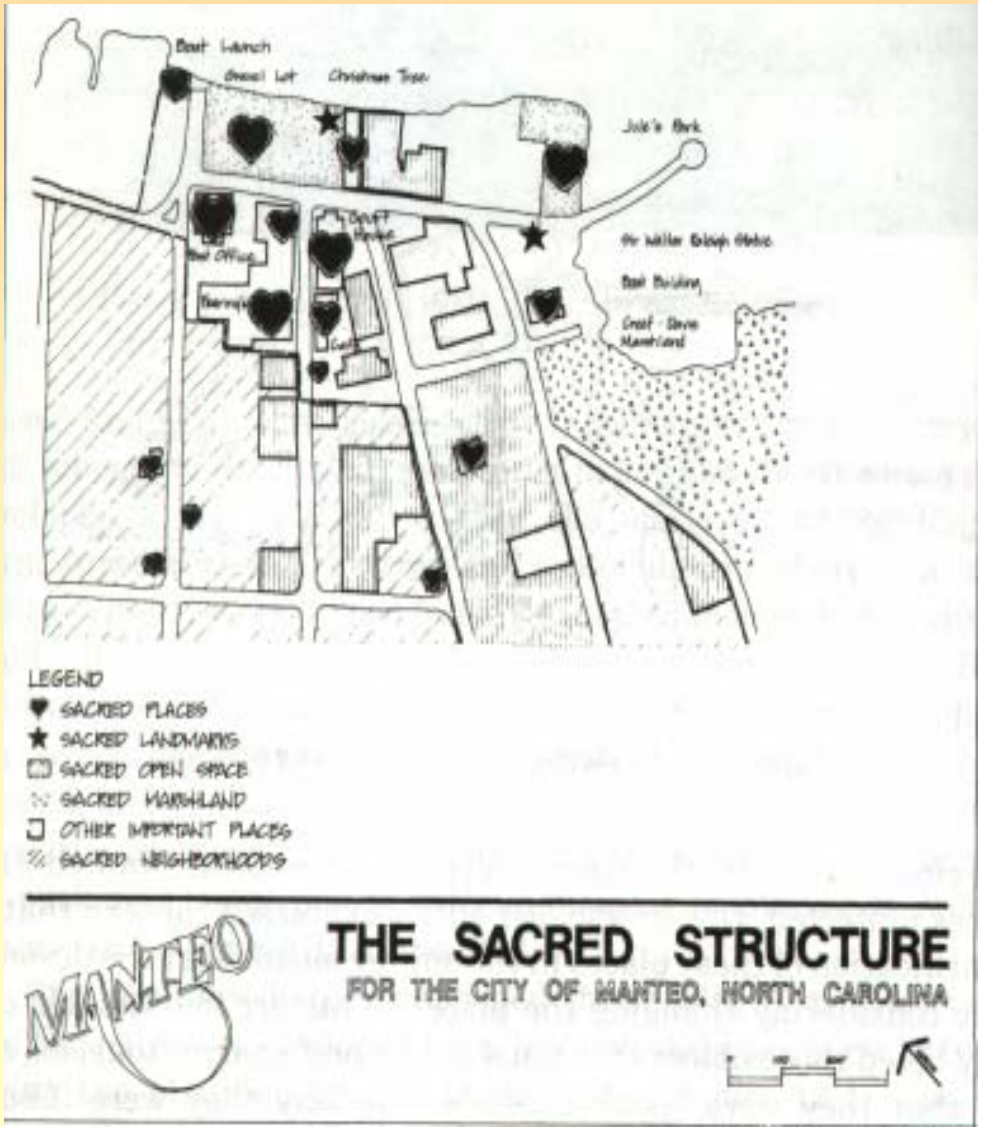
- El Proyecto:
  - Revitalizar el centro (town) de Manteo, North Carolina.

- Los desafíos:
  - Revitalizar el centro sin destruir sus aspectos característicos.
  - Detectar y potenciar las características que eran importantes para los habitantes de Manteo.
  - Establecer lineamientos of características (patrones) que deberían respetar las nuevas construcciones/remodelaciones.

The Spirit of Patterns: The Re-design of Manteo

El mapa de la Estructura Sagrada..

- Jules Park
- The Dutchess restaurant
- A gravel parking lot
- The post office
- The marshes (pantanos)
- Locally made street signs
- Fearing's soda shop
- The church
- The cemetery
- ... Etc.

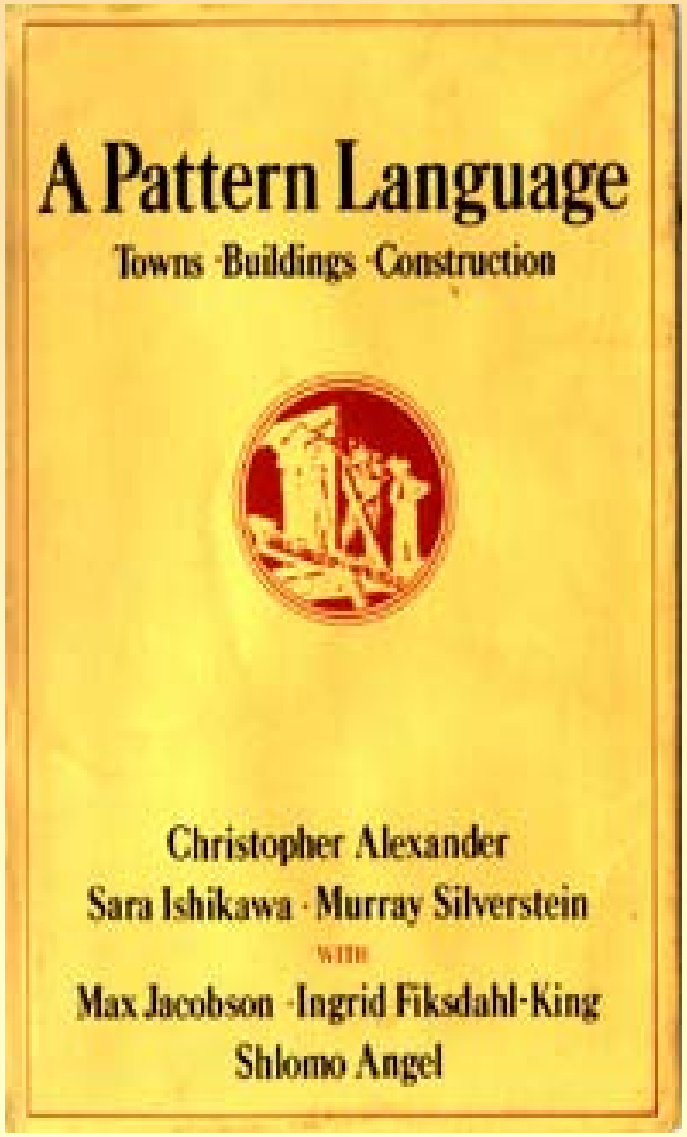


Ejemplo de Estos Patrones

Ejemplo de Estos Patrones

The Theory of Patterns: *A Pattern Language*

- ***Objetivo:***
  - Especificar calidad sin un nombre.
  - .... conocimiento reusable.
- Se obtuvieron 253 patrones.
- ... que tratan de conectar el comportamiento individual y social a las características del lugar.
- .... que tratan de rescatar los patrones de los lugares que son importantes para esta gente.



Investigación en Patrones de Arquitectura

- ¿Es posible formalizar esta comunicación para poder reusar el conocimiento experto?
- Premisas:
  - existen problemas recurrentes en el diseño de software,
  - las soluciones a estos problemas tienen ciertas propiedades invariantes (solución abstracta),
  - pueden capturarse los problemas y las soluciones.

Patrones de Arquitectura de Software

- Los patrones en general, y en particular los de arquitectura, son un intento de formalizar la comunicación y el reuso de la experiencia de diseño,
- capturan la experiencia probada de diseño de software,
- describen problemas recurrentes que surgen en determinados contextos,
- describen esquemas de soluciones probados,
- Son una herramienta para los no expertos,
- un paso más hacia la ingeniería de software:
  - permite que gente común haga cosas que antes requerían virtuosismo.

Patrones: el reuso de la experiencia

- Los diseñadores expertos manejan patrones recurrentes de clases y colaboraciones útiles ante determinados problemas.
- Los patrones resuelven problemas concretos y crean diseños más elegantes, flexibles y reutilizables.

Permiten la reutilización de la experiencia

¿Qué es un Patrón?

Un patrón de arquitectura de software es un esquema genérico probado para solucionar un problema particular, el cual es recurrente dentro de un cierto contexto. Este esquema se especifica describiendo los componentes, con sus responsabilidades y relaciones.

Características de los Patrones

- Atacan problemas recurrentes que ocurren en situaciones específicas y dan una solución.
  - Variabilidad de las interfaces.
- Documentan experiencias de diseño existentes y bien probadas.
  - Experiencia en el desarrollo de sistemas interactivos.
- Identifican y especifican abstracciones de alto nivel.
  - La tríada MVC es la que resuelve el problema.
- Proveen un vocabulario común y comprensible.
  - Todos entendemos cuando nos dicen “arquitectura MVC”.

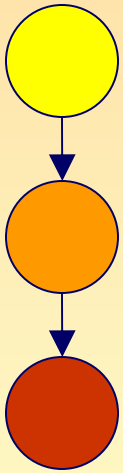
Más Características de los Patrones

- Son una forma de documentar la arquitectura del software.
- Facilitan la construcción de software con propiedades definidas (propiedades particulares).
  - Interfaces intercambiables y modelo reutilizable.
- Ayudan a construir arquitecturas heterogéneas y complejas.

Patrones en la Ingeniería de Software

- Pueden verse como bloques de construcción mentales, para tratar con distintos aspectos del diseño de software.
- Hay patrones de arquitectura, patrones de diseño, patrones de procesos, patrones de interfaces, “idioms”.
- No existen paradigmas o lenguajes específicos para implementar patrones de arquitectura, pero algunos proveen elementos útiles (orientación a objetos, polimorfismo, herencia, etc.).

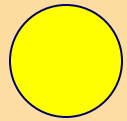
¿Cómo se especifica un patrón?



Patrón: Nombre del Patrón

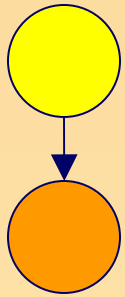
- **Contexto**
  - Situación de diseño que da lugar al problema.
- **Problema**
  - Conjunto de fuerzas que surgen del contexto.
- **Solución**
  - Configuración para balancear las fuerzas
    - Componentes y relaciones (estructura)
    - Comportamiento dinámico

Contexto



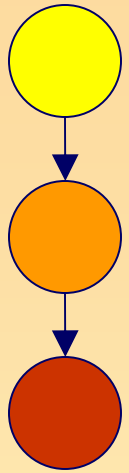
- Extiende la dicotomía problema-solución.
- Describe el escenario donde se da el problema.
- La descripción del contexto puede ser bastante general o muy específica. Por ejemplo:
  - “desarrollar software con una interfaz humano-computador”.
  - “implementar el mecanismo de cambio-propagación en MVC”.
- Es muy difícil definir completamente el contexto.
- Listar todas las situaciones en que el problema surge puede ser una alternativa.

Problema



- Describe el problema genérico que surge en el contexto especificado.
- Esencia: ¿cuál aspecto del problema debemos resolver?
  - El problema generalmente varía, pero la escencia se mantiene.
- Las *fuerzas* describen aspectos más específicos del problema con distintos puntos de vista, y hasta pueden contradecirse. Pueden ser:
  - requisitos de la solución
  - restricciones
  - propiedades deseables/indeseables

Solución



- Es la forma de balancear las fuerzas para resolver el problema.
- Dos aspectos:
  - Estructura estática - componentes y relaciones.
  - Comportamiento dinámico - forma de organización y colaboración entre las componentes.
- La solución no siempre toma en cuenta todas las fuerzas.
- Hay que establecer prioridades, si las fuerzas son contradictorias.
- Se debe establecer un esquema de soluciones y no algo completamente definido (demasiado especificado, y por ende, restrictivo).

Relación entre Patrones

- “Cada patrón depende de los patrones más pequeños que contiene y de los más grandes donde está contenido.”
- Distintos aspectos de un sistema pueden resolverse con distintos patrones.
- Existen variantes de ciertos patrones para situaciones especiales.

Descripción de Patrones

- Una descripción apropiada permite la comprensión, el análisis y la discusión del patrón.
- La definición del Contexto-Problema-Solución es un buen punto de partida.
- Los buenos nombres se convierten en jerga o modismos.
- Ejemplos, diagramas, escenarios y guías para la implementación.
- Discusión de beneficios y debilidades ayudan a tomar una decisión.

Descripción Completa de un Patrón

Ejemplo	Ejemplo conocido de la literatura
Nombre	Nombre descriptivo
Contexto	Situación en que surge el problema
Problema	Fuerzas que surgen en el contexto
Solución	Configuración que balancea las fuerzas
Estructura	Diagramas que describen la configuración
Dinámica	Descripción del comportamiento. Escenarios
Implementación	Guía para implementar el patrón
Resolución del ejemplo	Aplicación del patrón al ejemplo
Variantes	Alternativas para resolver el ejemplo
Usos conocidos	Descripción de problemas donde se aplica
Consecuencias	Beneficios y perjuicios

Categorías de Patrones Arquitectónicos

- Patrones Simples
  - Layers, Pipes-y-Filtros, Pizarrón, Repositorio
- Sistemas Distribuidos
  - Broker (Microkernel, Pipes-y-Filtros), CAGS, Cliente-Servidor
- Sistemas Interactivos
  - Modelo-View-Controlador, Presentación-Abstracción-Control
- Patrones Adaptables
  - MicroKernel, Reflexion

Patrones Arquitectónicos Simples

Patrones Simples

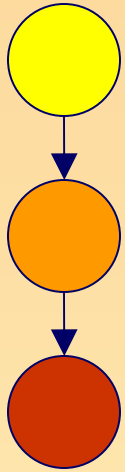
- ***Layers:*** Ayuda a estructurar aplicaciones que pueden ser descompuestas en grupos de subtareas con distintos niveles de abstracción (granularidad).
- ***Pipes-y-Filtros:*** Provee una estructura para sistemas que procesan datos de entrada. El procesamiento puede estar encapsulado en uno o más procesos (filtros).
- ***Pizarrón:*** Útil en problemas para los cuales no hay una solución conocida. Generalmente provee aproximaciones a la solución final.
- ***Repositorio:*** Ayuda a estructurar sistemas centrados en los datos, haciéndolos más flexibles y adaptables.

…. A todos ellos los hemos visto en clases pasadas, como modelos generales.

Layers

La arquitectura de layers o capas ayuda a estructurar aplicaciones que pueden descomponerse en grupos de subtareas con diferentes niveles de abstracción.

Layers: Contexto

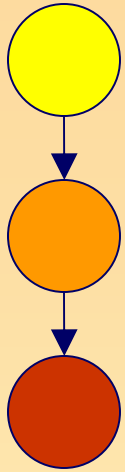


- Hay sistemas que utilizan distinta granularidad y naturaleza de servicios.

Layers: Problema

- Si los servicios no están bien organizados, el sistema podría tener problemas de mantenibilidad, adaptabilidad y escalabilidad.

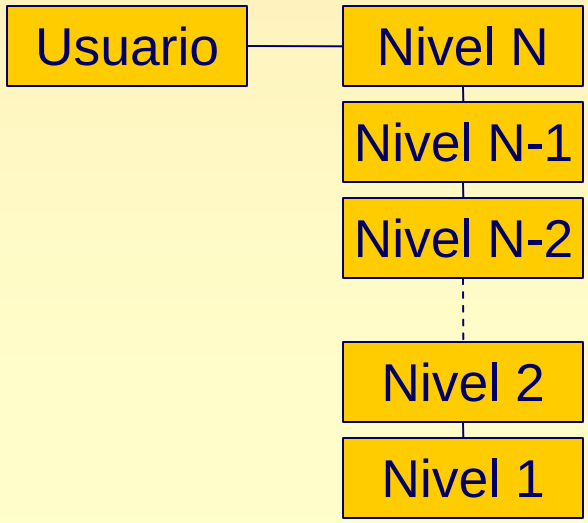
Layers: Solución



- Estructurar el sistema en un número apropiado de capas.
- Empezar con la capa con el nivel más bajo de abstracción.
- Poner la capa *J* sobre la *J-1* hasta alcanzar la capa *N*.
- Los servicios de *J* usan los servicios de *J-1*.
- No se requiere un orden en la implementación, ni ninguna sofisticación.
- Dentro de cada capa, las componentes tienen el mismo nivel de abstracción.

Layers: Estructura de la Solución

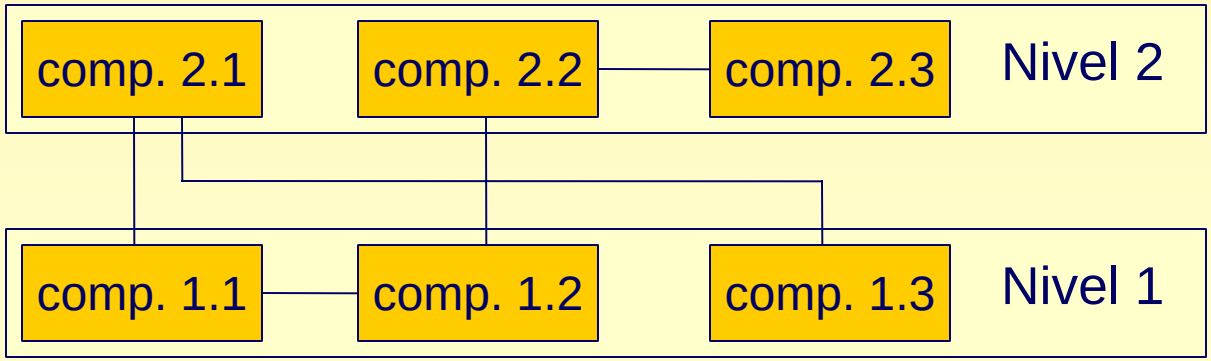
- Características principales de la estructura:
  - los servicios de la capa *J* se basan solamente en los de la capa *J-1*;
  - no existen otras dependencias entre las capas;
  - la estructura puede verse como una pila o una cebolla.



Usuario

Layers: Otra Estructura

- Cada capa puede ser una entidad compleja formada por distintas componentes.
- Una componente en la capa *J* puede llamar a:
  - otra componente en la capa *J*,
  - componentes en la capa *J-1* directamente,
  - una interfaz definida en la capa *J-1*.



Layers: Dinámica

- Escenario 1:
  - el usuario hace una *solicitud* a la capa *N*, ésta lo pasa a la capa *N-1*, y así sucesivamente hasta la capa 1 que efectivamente lo resuelve; eventualmente la respuesta vuelve a la capa *N* y al usuario;
  - generalmente una solicitud a la capa *J* se traduce en varias solicitudes a la capa *J-1* (nivel de abstracción).
- Escenario 2:
  - se inicia una comunicación bottom-up cuando la capa 1 detecta algún evento, por ejemplo cierto input;
  - esta *notificación* sube a través de las distintas capas.
- Escenario 3:
  - la comunicación sólo sucede entre ciertas capas.

Deben definirse el comportamiento de las componentes para todos los escenarios posibles.

En todo momento debe haber certeza respecto al comportamiento que tomarán las componentes.

Layers: Implementación

- Definir los niveles de abstracción para agrupar las tareas.
  - distancia desde el hardware o complejidad conceptual;
  - **Ej. Ajedrez:** piezas, movimientos básicos, tácticas (ej.: defensa Siciliana), estrategias globales del juego.
- Determinar el número de capas.
  - decisión de compromiso; no siempre coinciden con los criterios definidos para los niveles de abstracción.
- Designar y asignar tareas a las distintas capas.
  - la capa superior es el sistema tal como lo ve el usuario.

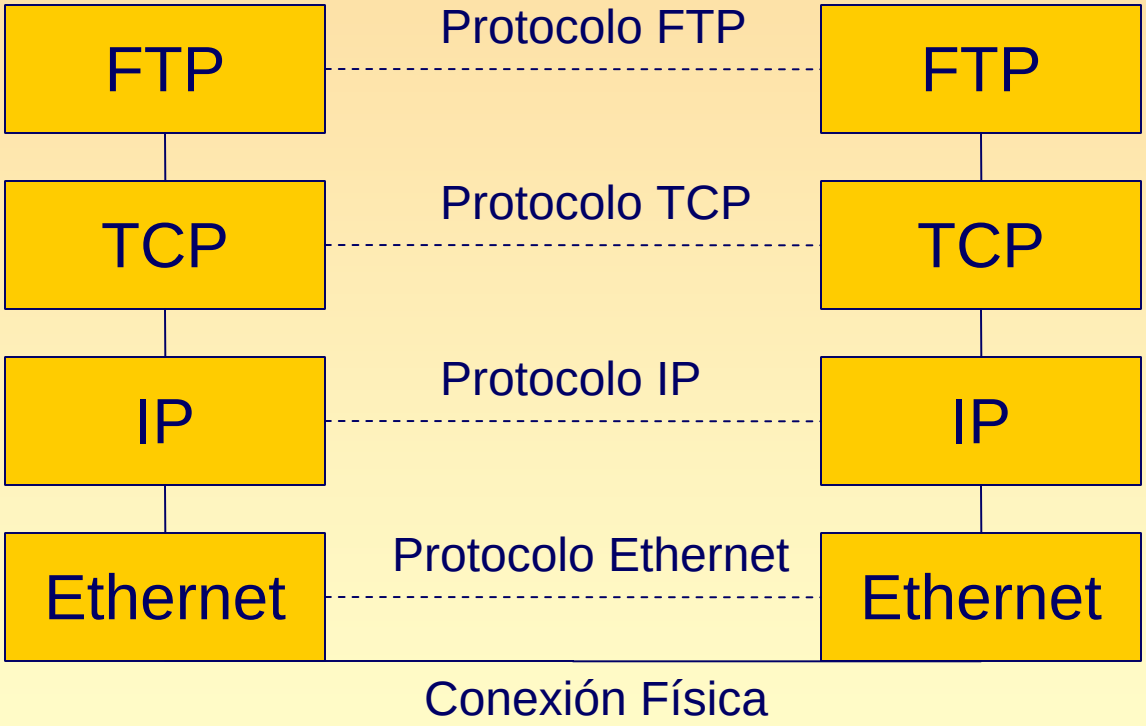
Layers: Implementación

- Especificar los servicios de cada capa.
  - es conveniente poner la funcionalidad dependiente de la aplicación en las capas superiores, y mantener las inferiores genéricas y sencillas.
- Refinar las capas iterando sobre los 4 pasos anteriores.
  - reorganizar las tareas de modo que sólo se invoquen servicios de la capa inmediatamente anterior.
- Especificar la interfaz de cada capa.
  - se incluyen todos los servicios ofrecidos en la interfaz y la capa se trata como una caja negra.

Layers: Implementación

- Definir la estructura de cada capa.
  - distintos patrones pueden usarse para la estructura interna.
- Especificar la comunicación entre las capas.
  - push, pull, llamadas a procedimientos, RPCs, mensajes.
- Desacoplar capas adyacentes.
  - la capa *J* no debe saber quien usa sus servicios.
- Diseñar una estrategia de manejo de errores.
  - los errores deben manejarse en la capa más baja posible.

Ejemplo: FTP de UNIX



Cada protocolo virtual es estricto.

Variantes

- Layers relajados:
  - cada capa puede acceder a servicios en cualquier capa inferior;
  - capas parcialmente opacas;
  - performance versus facilidad de mantenimiento.

Otras Aplicaciones

- Máquinas Virtuales.
- APIs.
- Sistemas de Información.

Layers: Consecuencias

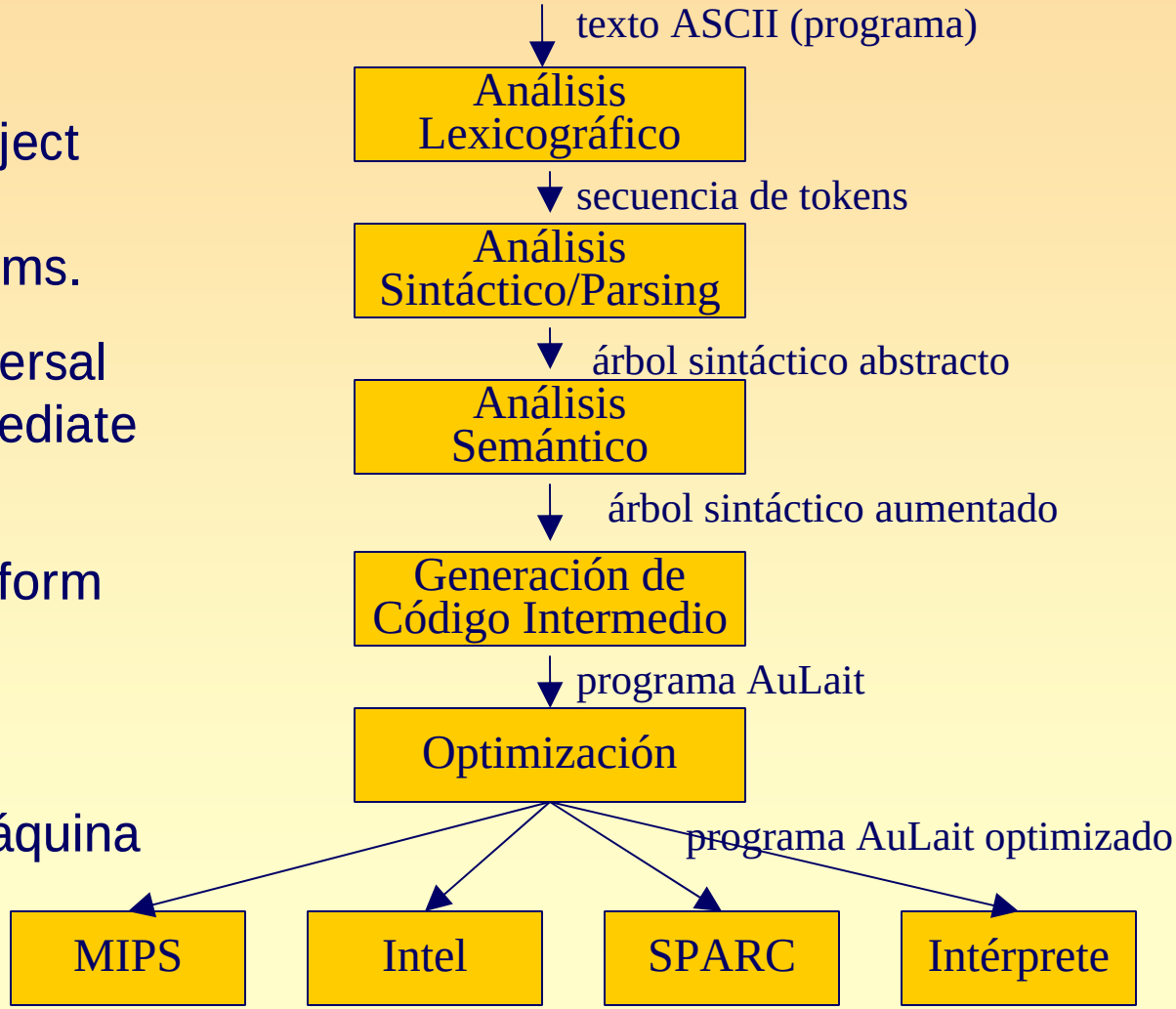
- Beneficios:
  - reutilización de niveles,
  - soporte para la estandarización,
  - los cambios en el código son locales a cada nivel.
- Desventajas:
  - cambios de comportamiento en cascada,
  - ineficiencia
  - excesivo trabajo,
  - difícil establecer la correcta granularidad de los niveles:
    - pocos niveles no explotan el potencial de reuso, portabilidad e intercambiabilidad,
    - muchos niveles crean complejidad e ineficiencia.

Pipes y Filtros

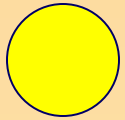
El patrón de arquitectura de pipes y filtros provee una estructura para procesar flujos de datos. Cada paso de procesamiento se encapsula en un filtro. Los datos se pasan usando los “pipes” entre filtros adyacentes. Recombinando los filtros pueden construirse distintas familias de sistemas relacionados.

Ejemplo: Compilador Portable de *MOCHA*

- *MOCHA*: Modular Object Computation with Hypothetical Algorithms.
- *AuLait*: Another Universal Language for Intermediate Translation.
- *CUP*: Concurrent Uniform Processor.
- *AuLait* corre en la máquina virtual *CUP*.

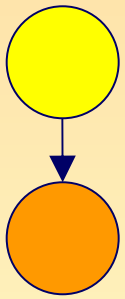


Pipes y Filtros: Contexto



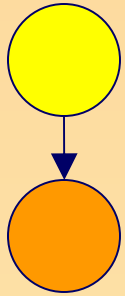
- Procesamiento de flujos de datos.

Pipes y Filtros: Problema



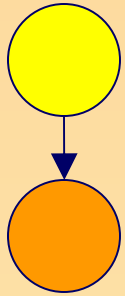
- La funcionalidad del sistema debe ser descompuesto en una serie de actividades.
- Las actividades transforman uno o más flujos de datos de entrada, en uno o más flujos de datos de salida en forma incremental.
- Cada actividad es un filtro.
- Los filtros se comunican solamente a través de pipes:
  - operación del sistema operativo que envía una secuencia de datos de un proceso a otro.

Pipes y Filtros: Problema



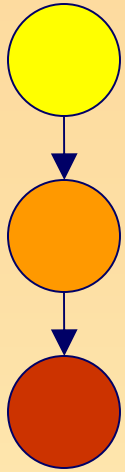
- El sistema es una secuencia de transformaciones de los datos.
- Las transformaciones son:
  - bien ordenadas,
  - sin estado interno,
  - independientes.

Pipes y Filtros: Problema



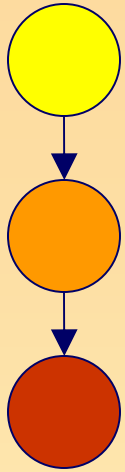
- Fuerzas:
  - es posible implementar cambios intercambiando algunos filtros;
  - pequeñas transformaciones son más fácilmente reutilizadas;
  - los datos pueden tener diferentes formatos.

Pipes y Filtros: Solución



- Involucra 4 elementos (según Mary Shaw):
  - Modelo de sistema: flujo de datos entre componentes;
  - Componentes: filtros (transformaciones, procesamiento local, asíncrono);
  - Conectores: pipes (streams);
  - Estructura de control: flujo de datos.

Pipes y Filtros: Solución



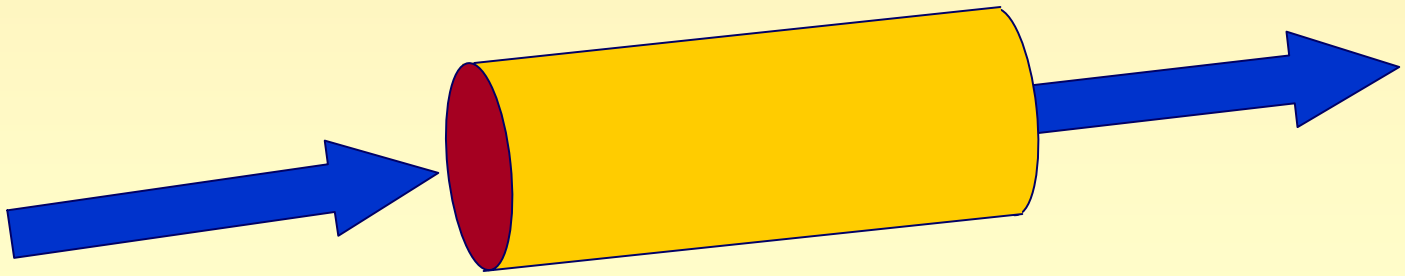
- Según Buschmann:
  - el sistema se divide en varios pasos secuenciales de procesamiento;
  - los pasos se conectan con flujos de datos;
  - cada filtro consume y procesa sus datos en forma incremental;
  - el origen de los datos, el destino y los filtros se conectan con “pipes” que implementan el flujo de datos entre los pasos de procesamiento.

Pipes y Filtros: Estructura

- Filtros:
  - unidades de procesamiento;
  - enriquece, refina y/o transforma datos;
  - el procesamiento se inicia:
    - el elemento siguiente solicita datos (pull),
    - el elemento anterior envía datos (push),
    - el filtro tiene un ciclo interno que solicita datos del filtro anterior y envía datos al siguiente (filtro activo);
  - los filtros no comparten estado;
  - los filtros no saben de la existencia o identidad de otros filtros.

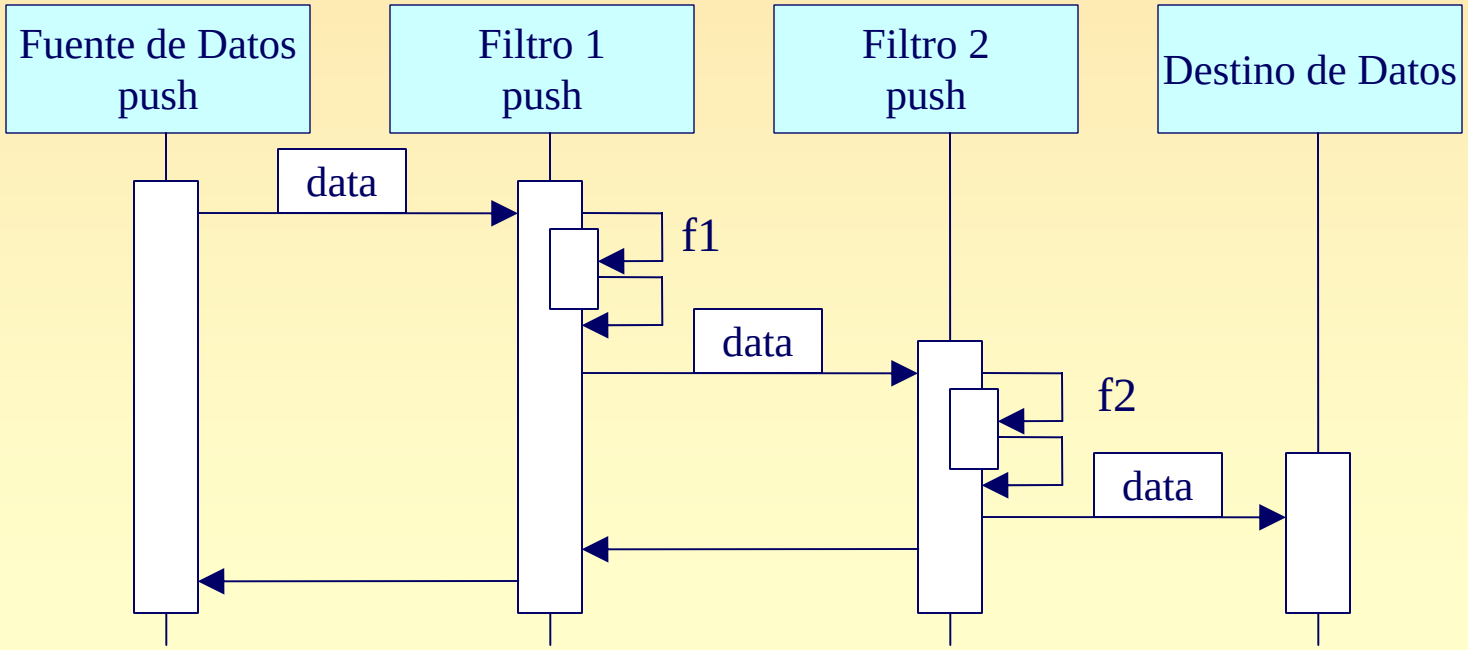
Pipes y Filtros: Estructura

- Pipes:
  - conecta origen-filtro, filtro-filtro, y filtro-destino;
  - transferencia de datos con un buffer *First-In-First-Out*.



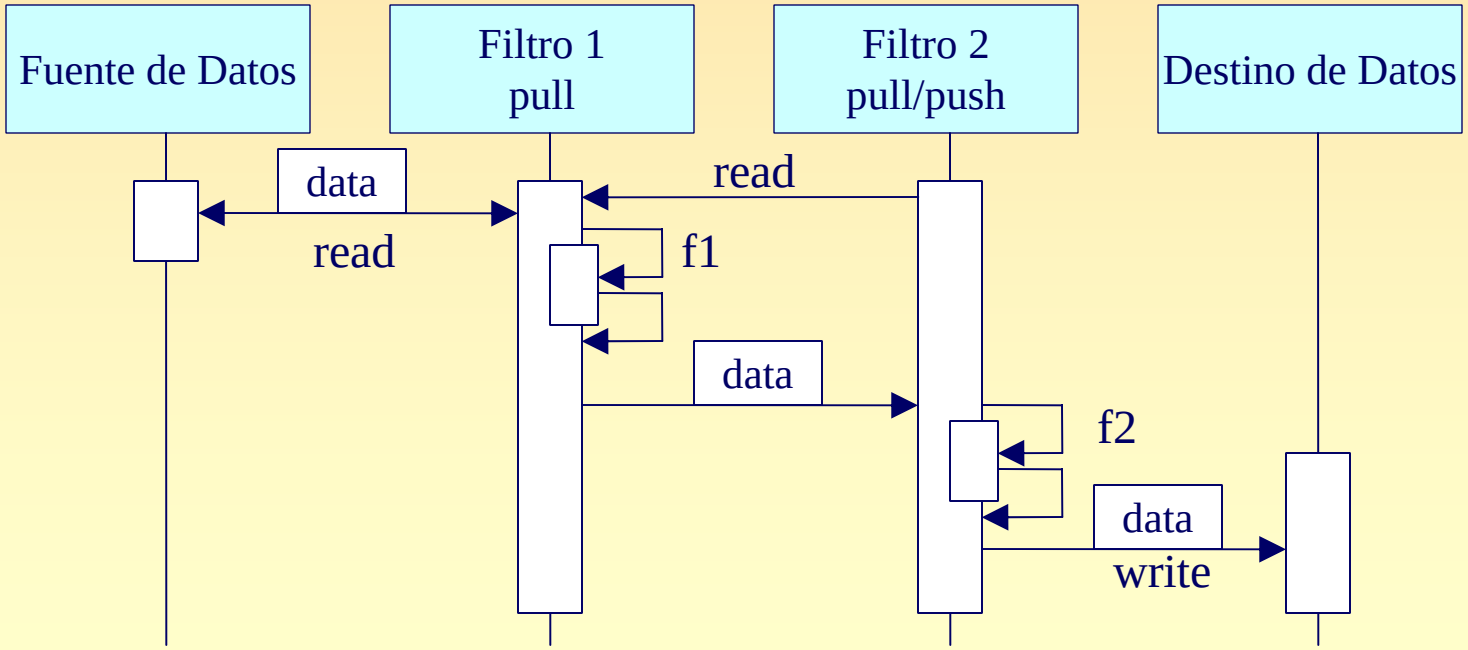
Pipes y Filtros: Dinámica

- Escenario 1: Push Pipeline.
  - La acción se inicia “empujando” los datos hacia el siguiente filtro.



Pipes y Filtros: Dinámica

- Escenario 2: Push/Pull Pipeline:
  - origen y destino de datos pasivos,
  - el filtro 2 juega el rol activo iniciando el proceso.

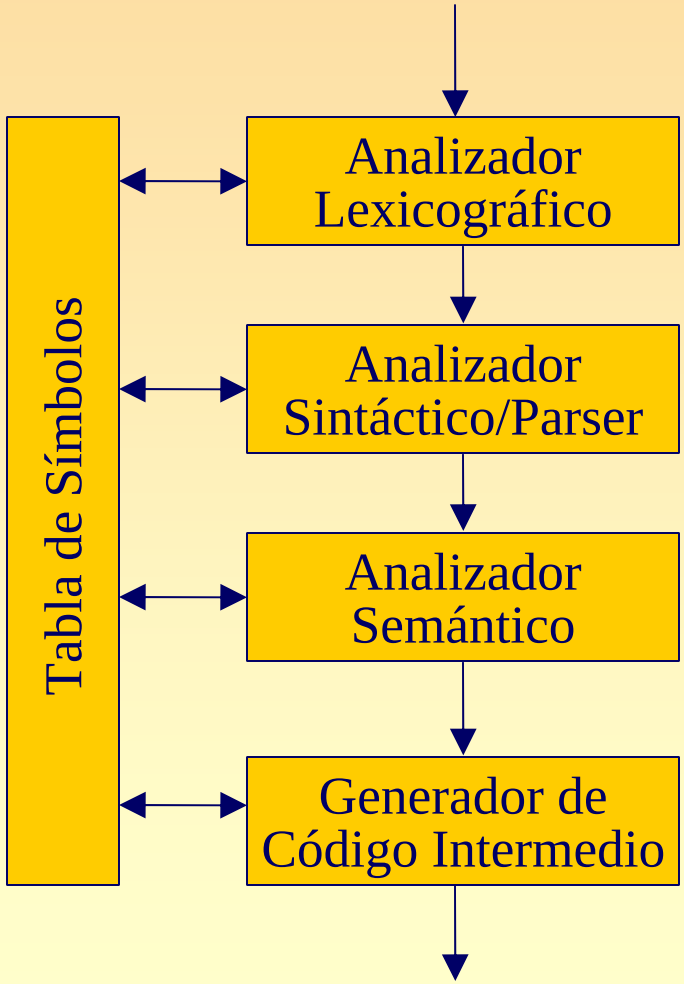


Pipes y Filtros: Implementación

- Dividir las tareas en una secuencia de pasos de procesamiento:
  - los procesos deben ser independientes y ordenados.
- Definir el formato de los datos transmitidos a través de cada pipe:
  - flexibilidad vs. performance.
- Decidir implementación de cada pipe:
  - determina la implementación de los filtros (activo o pasivo).
- Diseñar e implementar filtros.
- Diseñar política de errores.
- Instalar el sistema.

Ejemplo: Compilador MOCHA

- Los sucesivos filtros comparten un cierto estado: la tabla de símbolos.
- No es una arquitectura Pipes y Filtros estricta.
- La implementación es un único programa.



Pipes y Filtros: Consecuencias

- Beneficios:
  - los archivos intermedios no son necesarios,
  - flexibilidad,
  - reutilización de filtros,
  - rápida prototipación,
  - eficiencia con procesamiento paralelo.
- Desventajas:
  - compartir información de estado es caro y poco flexible,
  - ineficiencia por conversión de datos,
  - errores pueden implicar reiniciar el sistema.

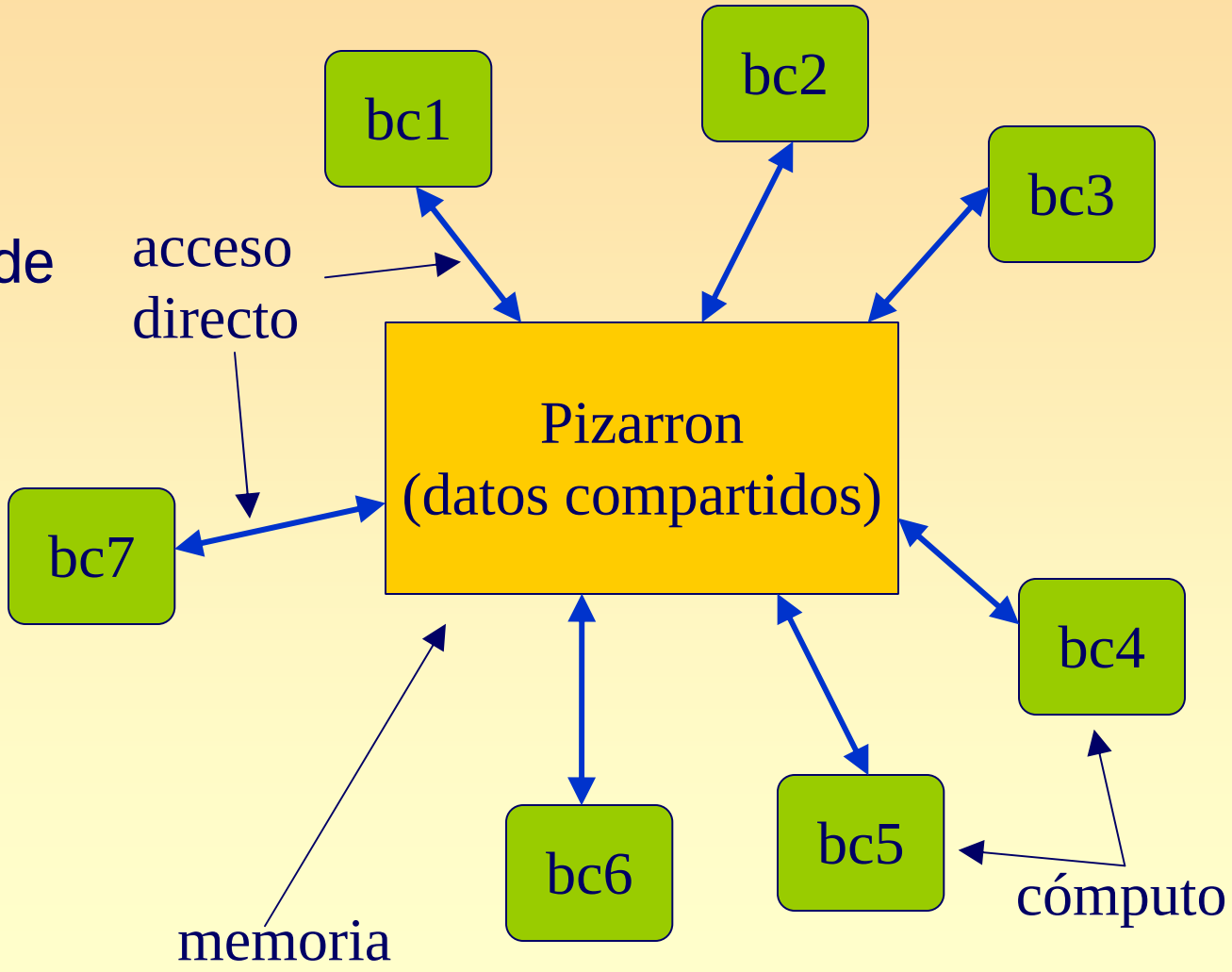
Pizarrón

El patrón de arquitectura de pizarrón es útil para problemas en que no se conoce una estrategia determinística para calcular la solución. Varios subsistemas especializados reunen su conocimiento para construir una solución parcial aproximada.

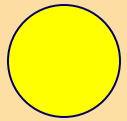
Variantes del Patrón: REPOSITORIO y CLIENTE-SERVIDOR

Ejemplos

- Visión artificial, reconocimiento de imágenes y voz.
- Inteligencia artificial.
- Sistemas colaborativos de apoyo a la toma de decisiones.

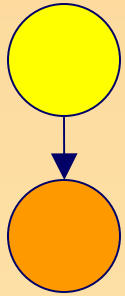


Pizarrón: Contexto



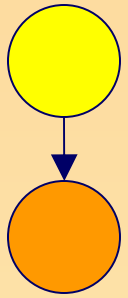
- Un dominio inmaduro en el cual no existe una solución conocida o factible de antemano.

Pizarrón: Problema



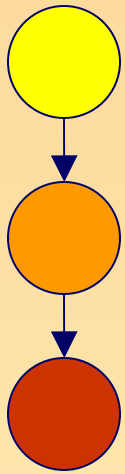
- La descomposición del problema abarca muchas áreas de especialidad.
- Cada solución parcial requiere distintas representaciones y paradigmas.
- El conocimiento es incierto o aproximado.

Pizarrón: Problema



- Fuerzas:
  - una búsqueda exhaustiva de soluciones no es factible,
  - los módulos deben ser intercambiables porque ninguno es seguro que obtenga la mejor solución,
  - existen algoritmos que obtienen soluciones parciales,
  - input, output y algoritmos usan datos con diferentes representaciones,
  - algunos algoritmos usan los resultados de los otros.

Pizarrón: Solución



- Colección de programas independientes que trabajan colaborativamente sobre datos compartidos.
- Cada programa se especializa en resolver parte del problema.
- Los programas no se comunican directamente sino a través de los datos.
- Existe un control central que coordina los programas dependiendo del estado de los datos (moderador): resolución oportuna de problemas.
- Se experimenta con soluciones parciales, se las combina y se las desecha.
- Los programas también pueden tomar iniciativa e interactuar con los datos.

Pizarrón: Estructura

- Pizarrón:
  - almacenamiento central de información,
  - vocabulario: conjunto de todos los datos que aparecen en el pizarrón.
- Bases de Conocimiento:
  - subsistemas independientes especializados,
  - escriben y leen del pizarrón,
  - parte de diagnóstico y parte de acción.
- Control:
  - monitorea el estado del sistema y decide la próxima acción,
  - las decisiones se basan en el progreso o costo del conocimiento,
  - detecta estados finales aceptables e insolubles.

Patrones Arquitectónicos para Sistemas Interactivos

Patrones para Sistemas Interactivos

- ***Modelo-View-Controlador***: divide a una aplicación interactiva en tres componentes: modelo, vistas y controladores.
- ***Presentación-Abstracción-Control:*** define al sistema como una jerarquía de agentes que coorperan entre sí para implementar la funcionalidad de la aplicación.

Modelo-View-Controlador

El patrón de arquitectura de modelo-view-controlador (MVC) divide una aplicación interactiva en tres partes. El modelo contiene los datos y la funcionalidad esencial. Las views despliegan la información al usuario. Los controladores manejan el input. Las views y los controladores juntos componen la interfaz con el usuario. El mecanismo de cambio-propagación asegura la consistencia de la interfaz con el modelo.

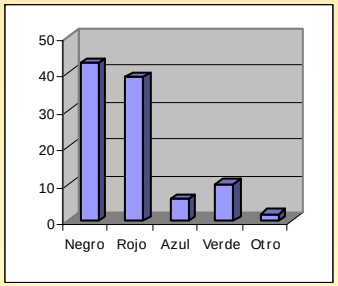
Modelo-Vista-Control (MVC)

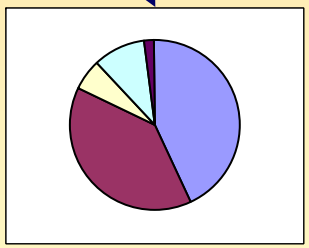
- Utilizado para construir interfaces de usuario en Smalltalk-80.
- Basado en tres tipos de objetos:
  - **Modelo**: objeto aplicación. Encapsula el núcleo funcional y los datos involucrados.
  - **Vistas**: presentación de información por pantalla (al usuario).
  - **Controlador**: define la forma en la que debe reaccionar la interfaz del usuario, frente a la entrada de datos (del usuario).
- Desacopla el modelo de las vistas.
- Hace a los sistemas más mantenibles, flexibles y adaptables.

Ejemplo: Sistema de Información

- Sistema de elecciones políticas.
- Los usuarios votan a través de una interfaz gráfica.
- La información se muestra con distintos formatos.
- Los cambios por votación deben reflejarse en forma inmediata.

Negro: 43%
Rojo: 39%
Azul: 6%
Verde: 10%
Otro: 2%





Negro	43
Rojo	39
Azul	6
Verde	10
Otro	2

- Debe poderse integrar otras formas de presentación de los resultados tales como la distribución de candidatos en el parlamento.
- Las presentaciones deben ser portables.

Modelo-Vista-Control (MVC)

Contexto: Aplicaciones interactivas con interfaces humano-computador cambiantes y flexibles.

Problema:

- Las interfaces de usuario son muy frecuentemente cambiadas.
- Cambios en la funcionalidad deben reflejarse en las interfaces.
- Puede haber interfaces a medida para ciertos usuarios.
- Diferentes paradigmas de interfaz:
  - digitar información,
  - seleccionar íconos.
- Construir un sistema monolítico es caro y difícil.

Modelo-Vista-Control (MVC)

Fuerzas:

- la misma información se presenta de distintas formas,
- cambios en los datos deben reflejarse en la interfaz inmediatamente,
- las interfaces deben modificarse fácilmente, ojalá durante la ejecución,
- distintas interfaces portables no deben afectar la operación esencial.

Modelo-Vista-Control (MVC)

Solución:

- MVC divide la aplicación en procesamiento, input y output.
- El modelo representa la funcionalidad y los datos esenciales, y es independiente de la representación en las interfaces.
- La view obtiene datos del modelo y los despliega para el usuario.
- Cada view tiene asociada un controlador. El controlador recibe eventos como input (movimientos del mouse, activación de botones) y los traduce a solicitudes de servicios del modelo o la view.
- El usuario interactúa con el modelo solamente a través de controladores.

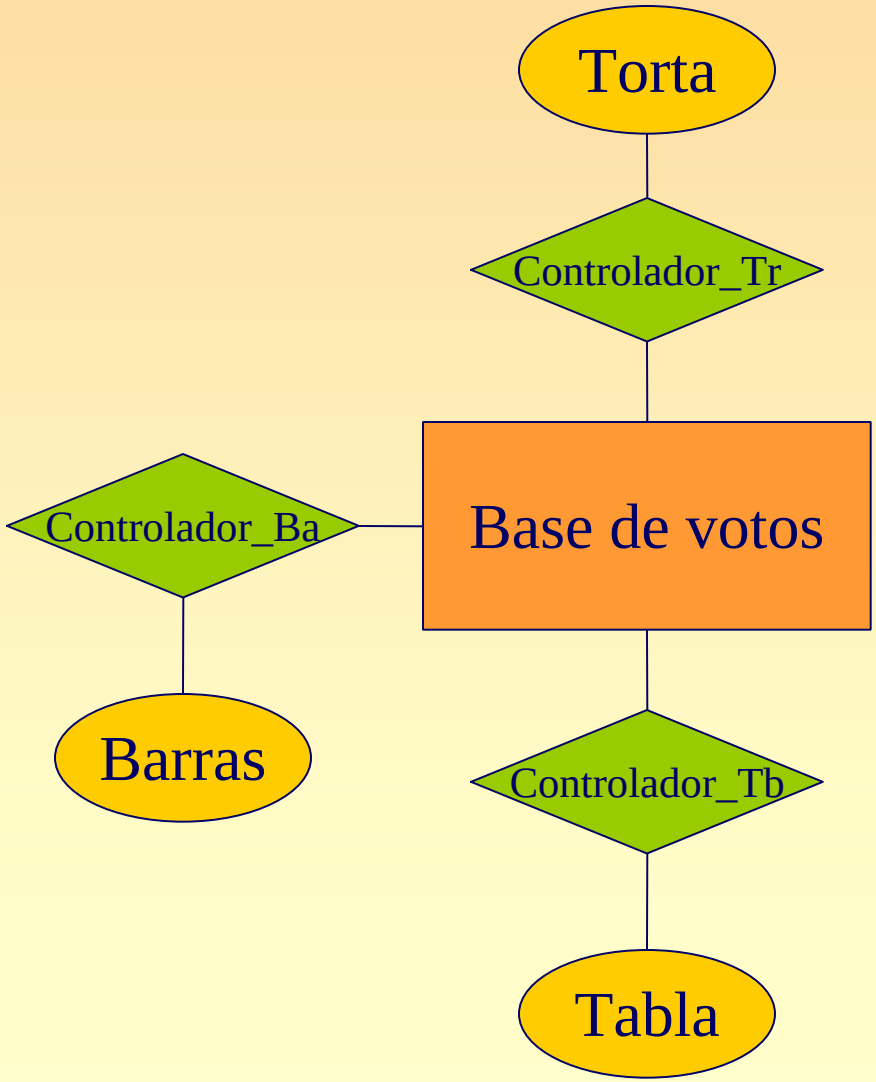
Modelo-Vista-Control (MVC)

Estructura:

- ***Modelo***
  - encapsula la información esencial y exporta procedimientos que realizan procesamiento específico de la aplicación;
  - provee funciones para que las views accedan a la información;
  - el mecanismo de cambio-propagación mantiene informados a las views y a los controladores dependientes.
- ***View***
  - despliega los datos para el usuario;
  - tienen procedimientos de actualización para recibir nuevos datos.
- ***Controlador***
  - existe un controlador para cada view;
  - recibe los inputs de una view como eventos y los interpreta.

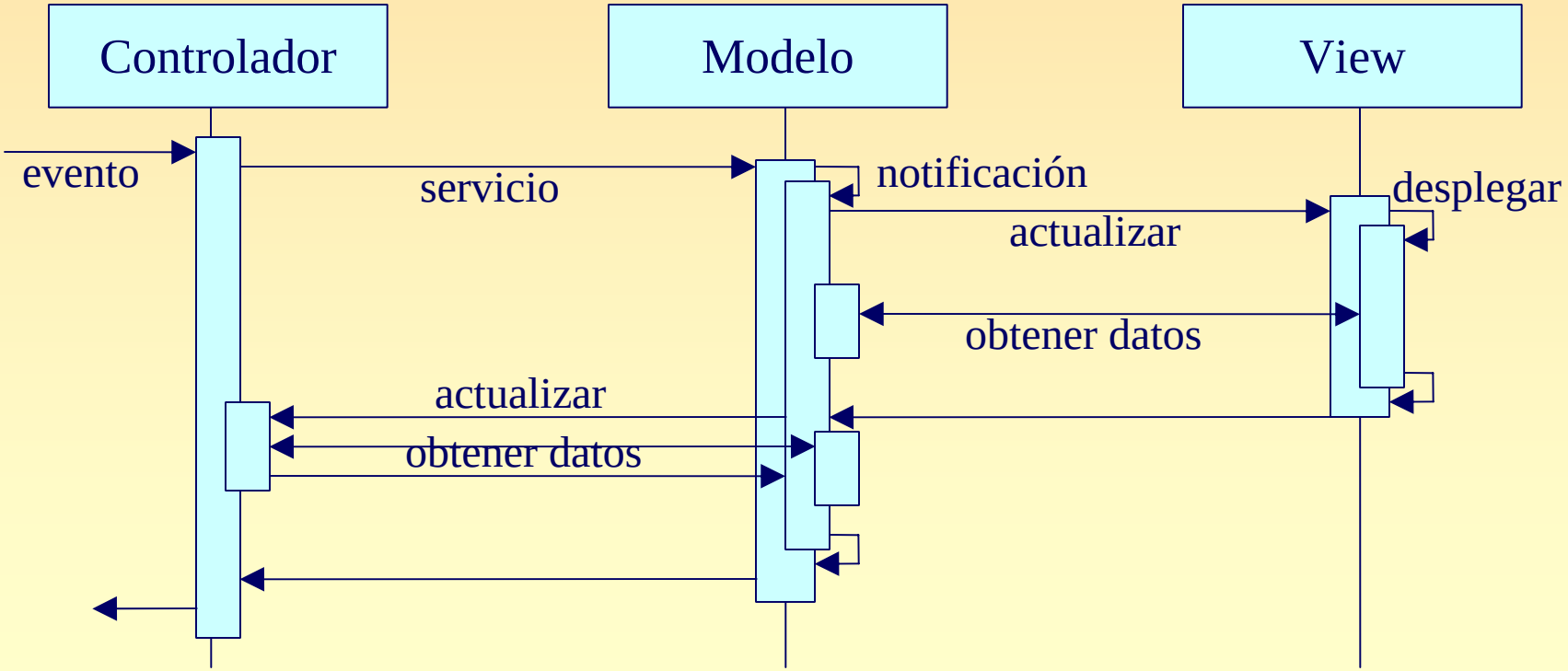
MVC: Estructura del Ejemplo

- El modelo guarda los votos acumulados para cada partido político y permite que las views accedan a los números de votos.
- Hay varias views: gráfico de barras, de torta o tabla.



MVC: Dinámica

- Escenario I: Input del usuario cambia el modelo y gatilla el mecanismo de cambio-propagación.



Modelo-Vista-Control (MVC)

Implementación:

- Separar la funcionalidad esencial de la interacción humano-computador.
- Implementar el mecanismo de cambio-propagación.
- Diseñar e implementar las views.
- Diseñar e implementar los controladores.
- Diseñar e implementar la relación entre views y controladores.

- Implementar:
  - La inicialización del MVC completo,
  - La creación de dinámica de views,
  - La controladores que capturan diferentes eventos,
  - La infraestructura jerárquica para views y controladores,
  - La más desacoplamiento de las dependencias del sistema.

Modelo-Vista-Control (MVC)

Consecuencias:

- Beneficios:
  - múltiples views para el mismo modelo,
  - views sincronizadas,
  - views y controladores intercambiables.
- Desventajas:
  - mayor complejidad,
  - excesivos cambios potenciales,
  - relación estrecha entre views y controladores,
  - gran acoplamiento entre views y controladores con el modelo,
  - ineficiencia en el acceso a los datos desde la view,
  - cambios inevitables a las views y controladores al portarlos,
  - difícil usar MVC con herramientas gráficas modernas.

Modelo-Vista-Control

- Utiliza los siguientes patrones:
  - **Observer**
  - **Composite**
  - **Strategy**
  - **Decorator**
  - **Factory method**
- ... Todos ellos son patrones de diseño.

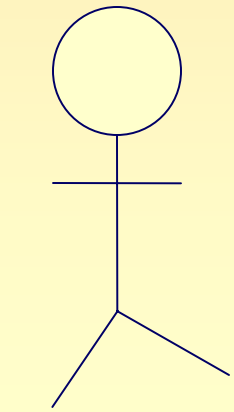
Presentación-Abstracción-Control (PAC)

El patrón de arquitectura PAC define una estructura jerárquica de agentes cooperativos. Cada agente es responsable de un aspecto específico de la funcionalidad de la aplicación y está compuesto por tres componentes: presentación-abstracción-control. Esta subdivisión separa los aspectos de HCI de los agentes, del núcleo funcional de cada uno y de los mecanismos de comunicación con otros agentes.

Presentación-Abstracción-Control (PAC)

Ejemplo: Consideremos un Sist. de Información Electoral.

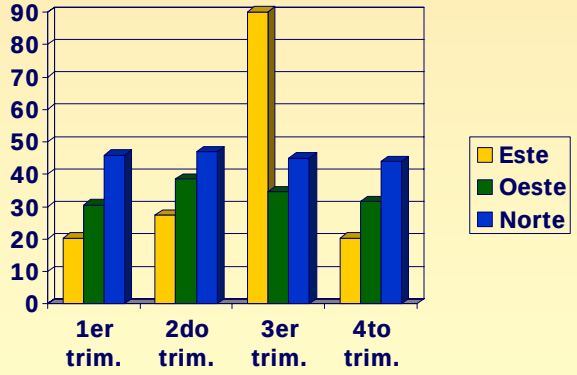
- Ofrece una hoja de cálculo para el ingreso de datos
- Ofrece varios tipos de tablas y de gráficos para representar los distintos resúmenes de la información almacenada.
- Los usuarios interactúan con el software a través de la interfaz gráfica.

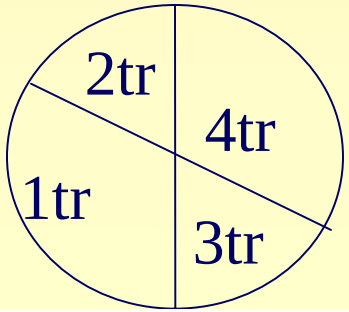


Usuario

	1tr.	2tr.	3tr.	4tr.
Este	20	25	90	20
Oeste	30	38	32	32
Norte	47	47	45	45

Hoja de Cálculo





Presentación-Abstracción-Control (PAC)

Ejemplo (cont.):

- Diferentes versiones, adaptan la interfaz de usuario a necesidades específicas, por ej., para ver las bancas del parlamento en función de los partidos políticos.

Contexto: Desarrollo de sistemas interactivos con la ayuda de agentes.

Problema: Cómo organizar el conjunto de agentes que forman parte de un sistema interactivo, para que funcionen en forma integrada.

Presentación-Abstracción-Control

Fuerzas:

- Los agentes normalmente mantienen su estado y sus datos, sin embargo tienen que cooperar y trabajar en conjunto.
- Los agentes proveen su propia interfaz de usuario pues cada uno tiene una cierta interacción prevista con el usuario. Sin embargo, las modalidades de interacción deben ser similares para que la HCI del producto sea homogénea.
- Separar la presentación, de la funcionalidad, para que los cambios en la interfaz no afecten demasiado al resto del sistema. De todos modos la presentación y la funcionalidad deben tener parámetros de acoplamiento aceptables.

Presentación-Abstracción-Control

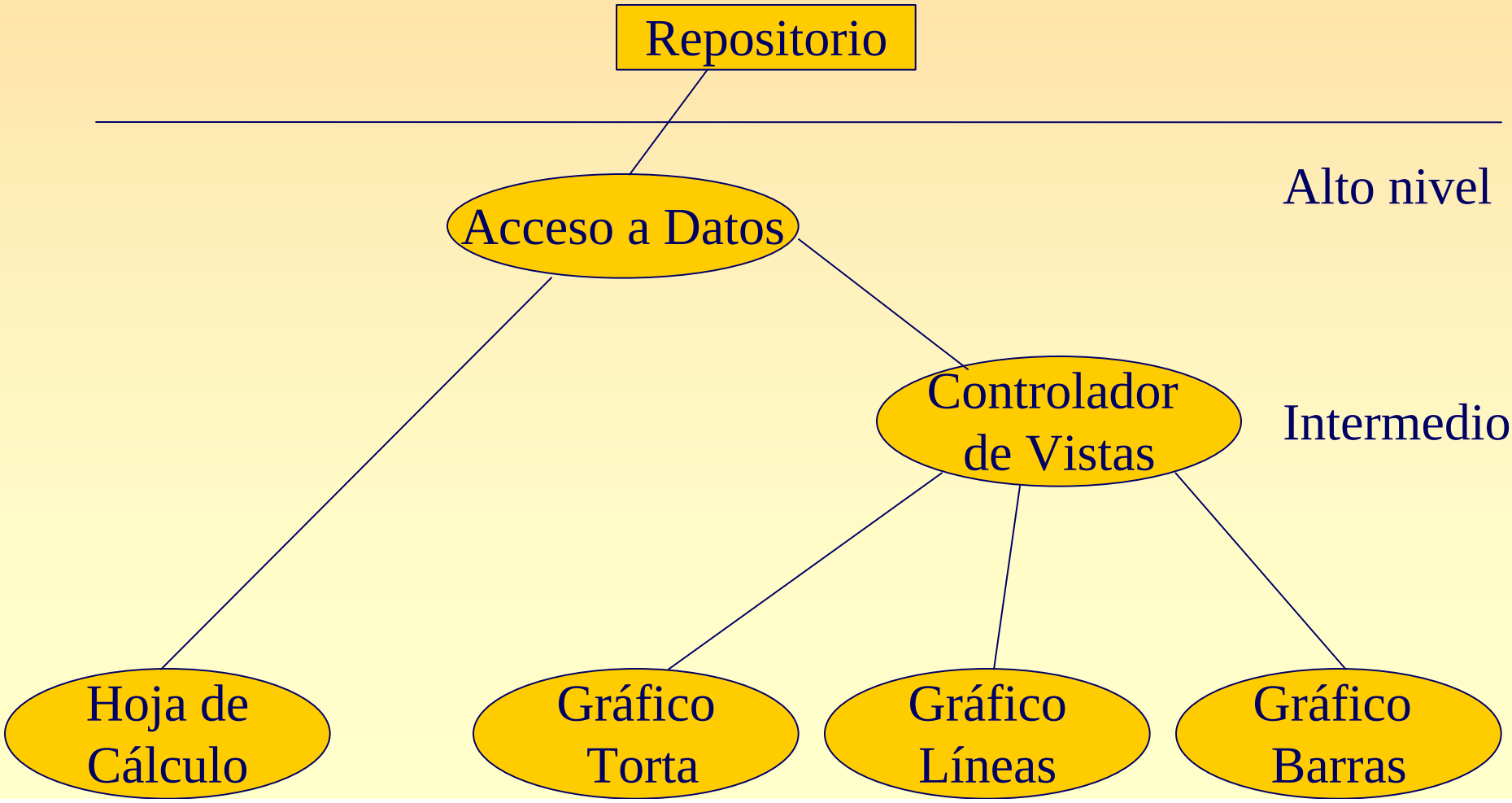
Solución:

- Organizar la estructura del sistema, jerárquicamente, según 3 niveles de agentes: *alto nivel*, *intermedio* y *bajo nivel*.
- Los ***agentes de alto nivel*** proveen el núcleo de la funcionalidad del sistema. Este tipo de agente es quien coordina y estructura la funcionalidad del sistema (menúes, barras de herramientas, etc).
- Los ***agentes de bajo nivel*** representan conceptos autocontenidos, sobre los cuales los usuarios del sistema actúan. Típicamente estos agentes soportan las operaciones de los usuarios.
- Los ***agentes intermedios*** representan combinaciones o relaciones entre agentes de bajo nivel. Por ejemplo, este tipo de agentes puede representar diversas vistas sobre los datos.

Presentación-Abstracción-Control

Solución:

- *Ejemplo:* veamos un sistema de información electoral.



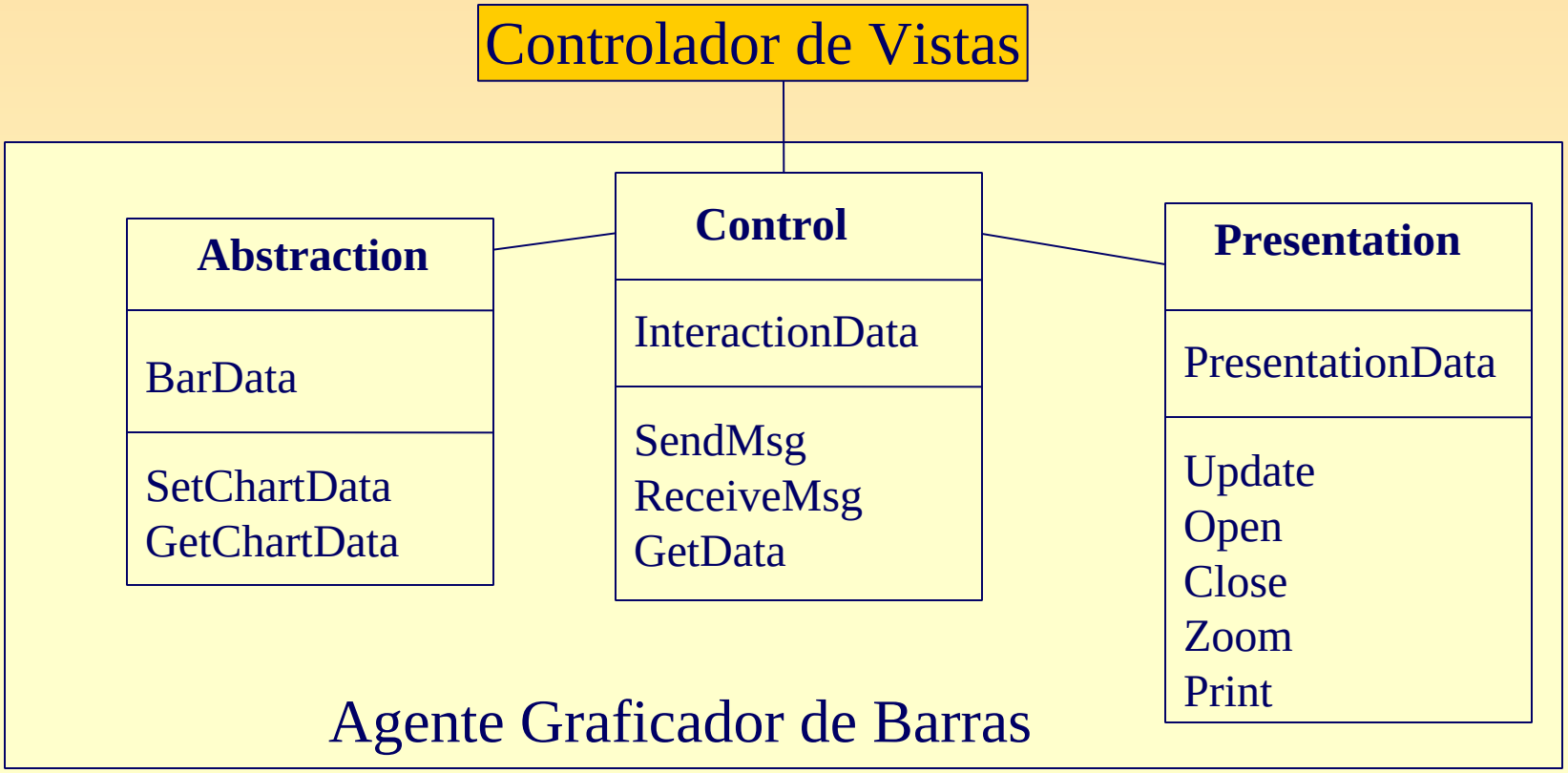
Presentación-Abstracción-Control

Solución:

- Cada uno de los agentes es responsable de una parte de la funcionalidad de la aplicación y contiene 3 componentes: presentación, abstracción y control.
- La ***presentación*** provee el aspecto visible del agente.
- La ***abstracción*** provee el modelo de datos del agente y las operaciones para operar con estos datos.
- El ***control*** conecta los componentes de presentación y de abstracción, y le agrega la funcionalidad necesaria para comunicarse con otros agentes.

Presentación-Abstracción-Control

Estructura: cada agente representado en la jerarquía tiene la siguiente estructura.



Presentación-Abstracción-Control

Dinámica: para cada escenario que sea representativo, es necesario definir la dinámica de las clases que forman parte de la estructura. Definir al menos un par de escenarios.

Consecuencias:

- *Beneficios*: Separación de conceptos, soporte para el cambio y la extensión, soporte multitarea.
- *Complicaciones:* Incrementa la complejidad del sistema, presenta problemas de eficiencia y de aplicabilidad, involucran componentes de control complejos.

Patrones para Sistemas Adaptables

Patrones para Sistemas Adaptables

- ***MicroKernel**:* Es usado en sistemas de software que deben adaptarse a los cambios en los requisitos. Este separa el núcleo funcional, la funcionalidad extendida y los aspectos relativos al cliente.
- ***Reflexion:*** Provee un mecanismo para cambiar la estructura y el comportamiento de un sistema de software, en forma dinámica. Este patrón divide a una aplicación en dos partes: un meta nivel que provee información acerca de las propiedades del subsistema seleccionado, y un nivel base que provee la lógica de la aplicación.

MicroKernel

- Ejemplos de sistemas MicroKernel son los sistemas operativos: NextSTEP, UNIX System V, MS Windows, OS/2 Warp.

Contexto: Desarrollo de aplicaciones que utilizan interfaces de programación similares, sobre las cuales construye un núcleo de funcionalidad similar.

Problema: El desarrollo de software de dominio específico es una tarea no trivial (Sist. Operativos, GUIs, etc). Los tiempos de vida de estos sistemas son largos, y tienen que incorporar los cambios tecnológicos con el menor costos (y tiempo) posible.

Fuerzas:

- La plataforma de la aplicación debe contemplar los cambios tecnológicos tanto en hardware como en software. Sin embargo, considerar todas las posibilidades podría elevar el costo innecesariamente.
- La plataforma de la aplicación debe ser portable, extensible y adaptable, para permitir la integración de las tecnologías emergentes. Estas tecnologías pueden ser actuales o futuras.

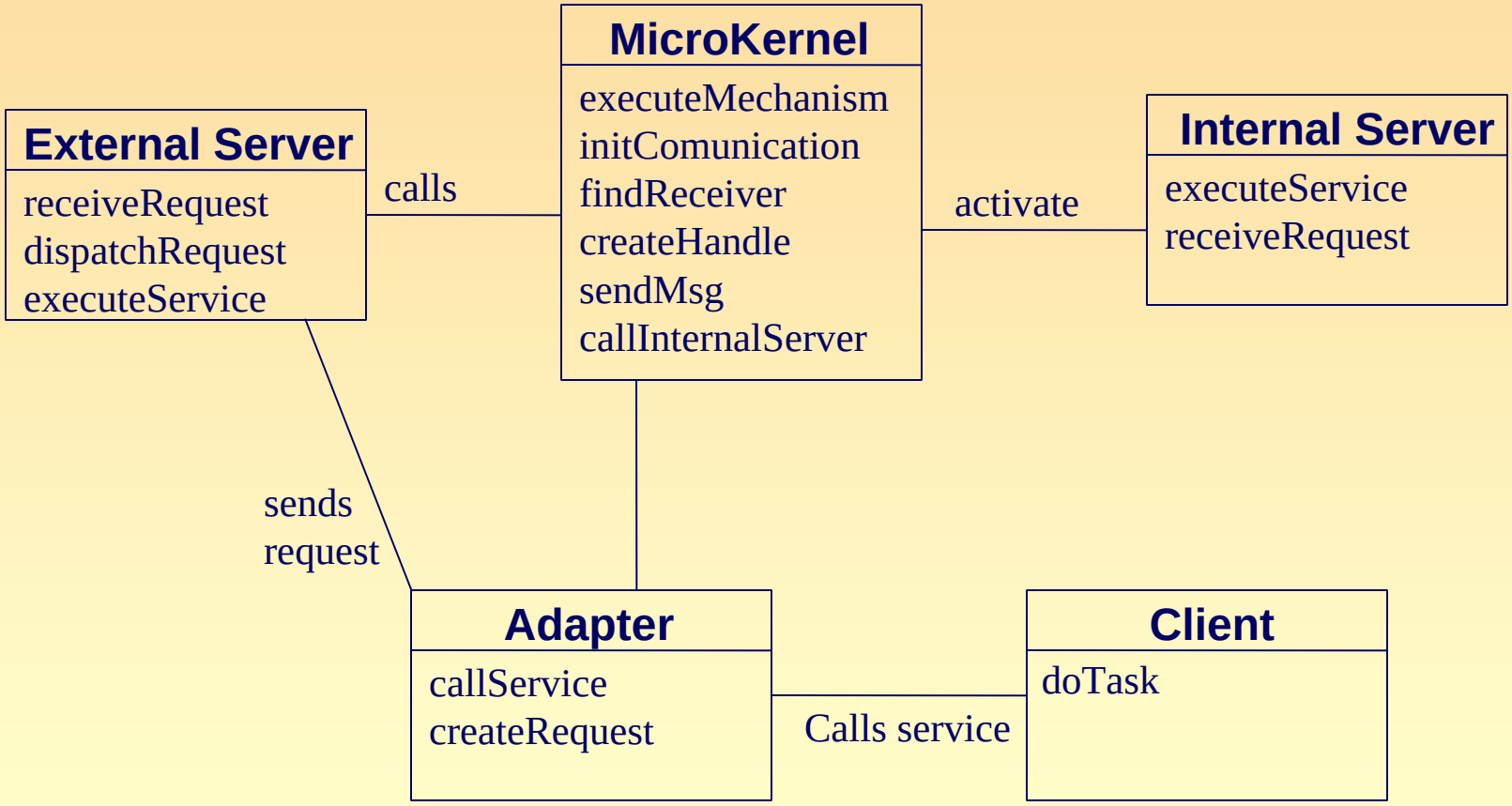
MicroKernel

Solución:

- Encapsular los servicios fundamentales de la plataforma de aplicación, en un componente microkernel. El microkernel incluye la funcionalidad que permite a otros componentes, ejecutar en forma separada (como proceso) y comunicarse entre ellos. Este es conocido como *internal server*.
- Encapsular los servicios periféricos al núcleo en subsistemas que agrupan servicio similares. Estos son conocidos como external servers.
- Los clientes se comunican con los external servers usando las facilidades de comunicación provistas por el microkernel.

MicroKernel

Estructura:



MicroKernel

Dinámica: Idem a patrones anteriores.

Consecuencias:

- *Beneficios*: Portabilidad, flexibilidad, escalabilidad, confiabilidad, transparencia.
- *Problemas*: Performance, Complejidad del diseño e implementación.

Recomendaciones Generales

- *Desempeño*.
  - Si el desempeño es un requisito crítico, la arquitectura debe estar diseñada para albergar las operaciones críticas, dentro de un número reducido de subsistemas (menos cambios de contexto).
  - Ojalá con poca comunicación entre ellos.
  - Esto significa utilizar componentes de grano grueso, de esa manera habrá menos componentes en el sistema.
- *Seguridad.*
  - Si la seguridad es un requisito crítico, se debe utilizar una arquitectura estratificada (por capas).
  - Los recursos más críticos deben alojarse en las capas más inferiores (internas).
  - Cada capa debe proveer un mecanismo de validación, de acuerdo a la información que ésta maneja.

Recomendaciones Generales

- *Protección (de operaciones)*.
  - Si la protección es un requisito crítico, la arquitectura debe estar diseñada de tal forma que las operaciones relacionadas con la protección, se localicen en único subsistema o en un conjunto muy reducido de estos.
  - Esto reduce los costos y los problemas de validación, y hace posible crear sistemas de protección relacionados.
- *Disponibilidad*.
  - Si la disponibilidad es un requisito crítico, como parte de la arquitectura se deben incluir componentes redundantes.
  - Estos podrán reemplazar a los componentes con problemas, en caso de fallas.

Recomendaciones Generales

- *Mantenibilidad*.
  - Si la mantenibilidad es un requisito crítico, la arquitectura debe estar diseñada utilizando componentes autocontenidos de grano fino.
  - Estos componentes pueden ser módulos o componentes de software bien definidos.
  - Estos podrán cambiarse o reemplazarse con facilidad, y con un mínimo efecto sobre el resto del sistema.
  - Los productores de datos deben estar separados de los consumidores.
  - Las estructuras de datos compartidas deben evitarse.
  - La circulación de órdenes de control entre módulos o subsistemas, también debe evitarse.

Conclusiones

- Hay arquitecturas de son mejores que otras, dependiendo del escenario de trabajo (dominio + infraestructura previa), del tipo de software a desarrollar, y de lo que se pretende obtener.
- Los patrones arquitectónicos brindan una sugerencia (genérica) acerca de cómo resolver algunos de los problemas arquitectónicos mas comunes.
- Los patrones arquitectónicos no son la salvación de nadie, pero generalmente ayudan....
- Se pueden utilizar mezclas de patrones, y adaptaciones de ellos.