Ingeniería de software[7]

Yadran Eterovic S. (yadran@ing.puc.cl)

Arquitectura (lógica) de software

Como dijimos en la clase “Diseño y arquitectura de software”, la **arquitectura** es diseño, pero a mayor escala

El diseño de un sistema o aplicación a esta escala se basa, p.ej., en varias *capas arquitectónicas*—**arquitectura por capas** o **estratificada** (próx. diap.)

Distinguimos la **arquitectura lógica** de un sistema:

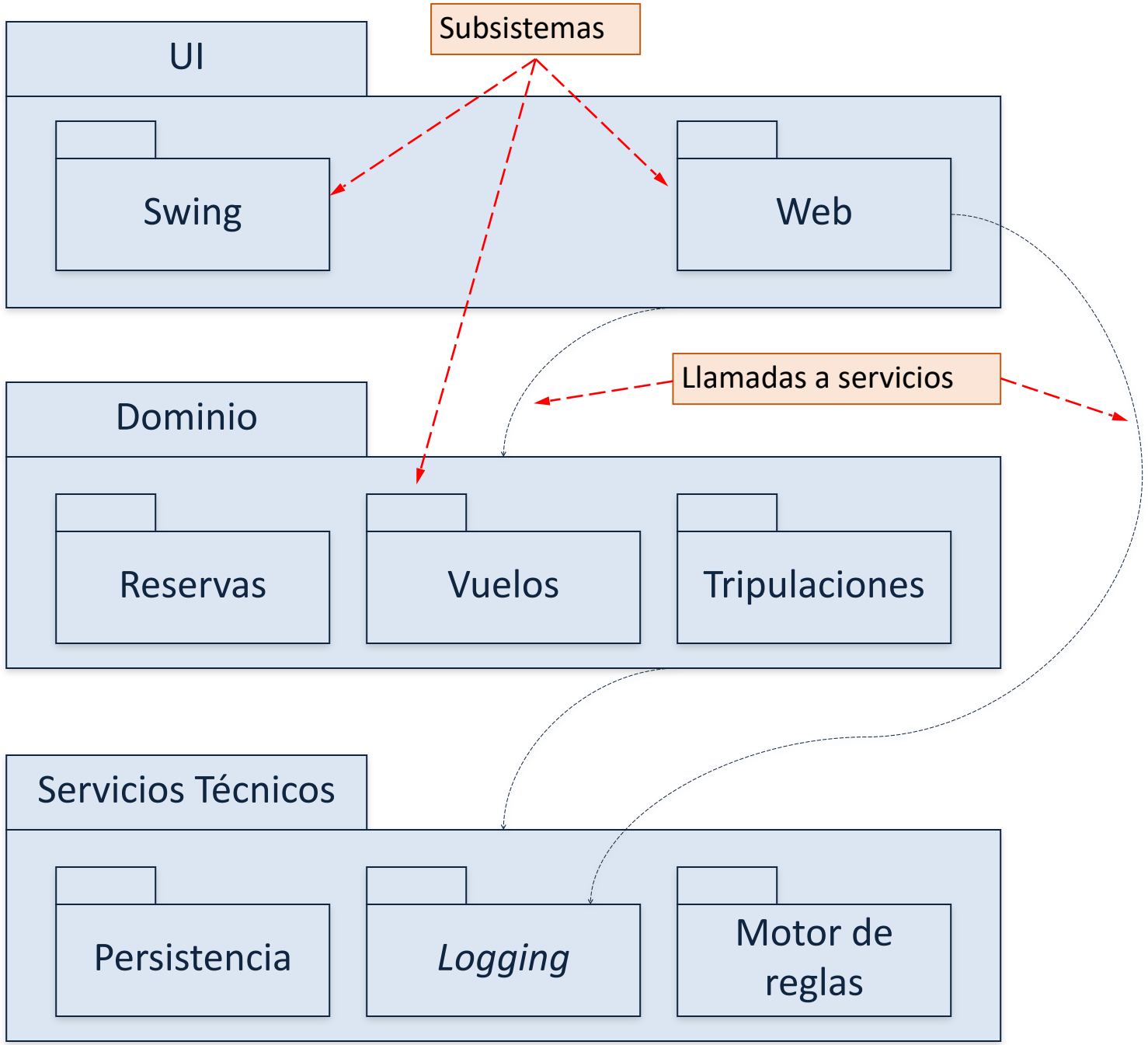
- la organización a gran escala de las clases del software

… de la **arquitectura física**:

- cómo se distribuirán los elementos de la arquitectura lógica entre los diversos procesos del sistema operativo o entre los distintos computadores físicos de una red

Nuestra preocupación en la clase de hoy

Las capas de más “arriba” (p.ej., la interfaz de usuario, o UI) son más específicas a la aplicación y llaman a los servicios de las de más abajo



Capa: Agrupación de clases, paquetes o subsistemas que, de manera cohesiva, tiene responsabilidad por un aspecto importante del sistema

Las capas de más “abajo” son servicios generales y de bajo nivel

Capas típicas en sistemas de información

UI (Presentación o Vista):

- más específicos a la aplicación

Aplicación (*Workflow*, Proceso, Controlador de Aplicación)

Dominio (Negocio, Lógica de Aplicación, Modelo)

Infraestructura de Negocio (Servicios de Negocio de Bajo Nivel)

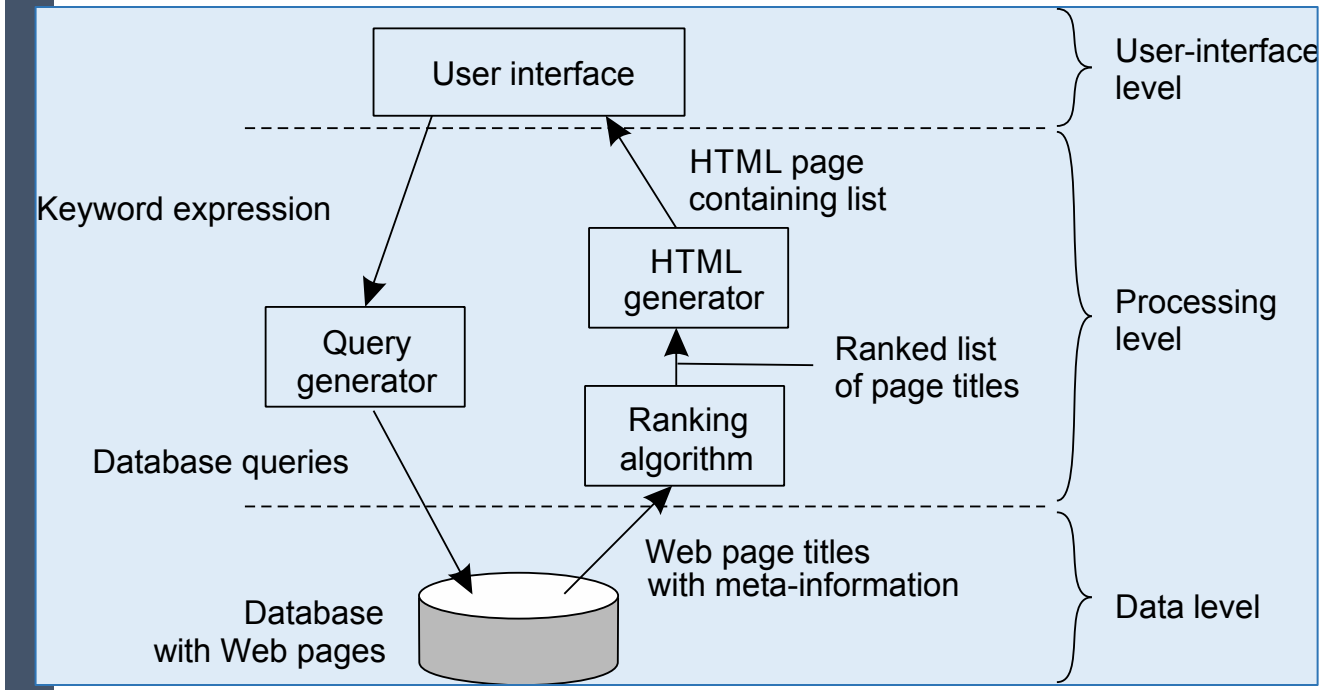
Servicios Técnicos (Infraestructura Técnica, Servicios Técnicos de Alto Nivel)

Fundación (Servicios *Core*, Servicios Base, Servicios/Infraestructura Técnicos de Bajo Nivel):

- rango amplio de aplicabilidad

La estratificación lógica de las aplicaciones, p.ej., diferenciando entre tres niveles lógicos que siguen un estilo arquitectónico estratificado para permitir acceso a bases de datos:

- el nivel de la interfaz de la aplicación, que maneja la interacción con el usuario o con una aplicación externa
- el nivel de procesamiento, que contiene la funcionalidad central de la aplicación y que varía mucho de una aplicación a otra
- el nivel de datos, que opera sobre una base de datos o sistema de archivos **persistente**



el usuario escribe un string de palabras clave, y recibe de vuelta el despliegue de una lista de títulos de páginas Web

programa que transforma las palabras clave en una o más consultas a la base de datos, que luego coloca los resultados en una lista ranqueada, y que finalmente transforma la lista a una serie de páginas HTML

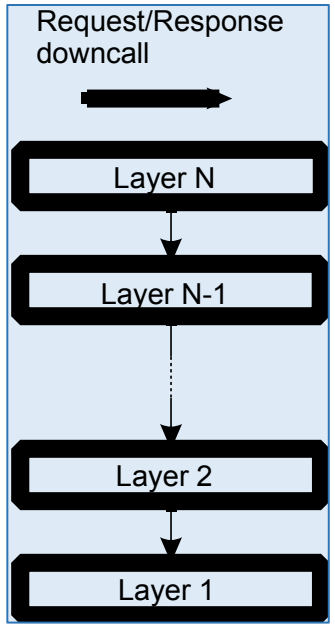
páginas Web *prefetched* e indexadas

Motor de búsqueda en Internet

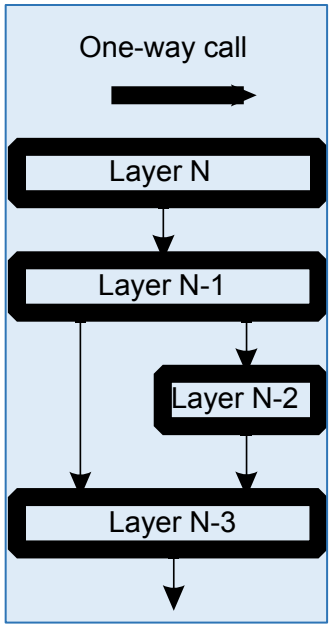
Arquitecturas estratificadas o de capas

En general, un componente en la capa L_j sólo puede llamar (*downcall*) a componentes en capas L_i de más abajo ($i < j$)

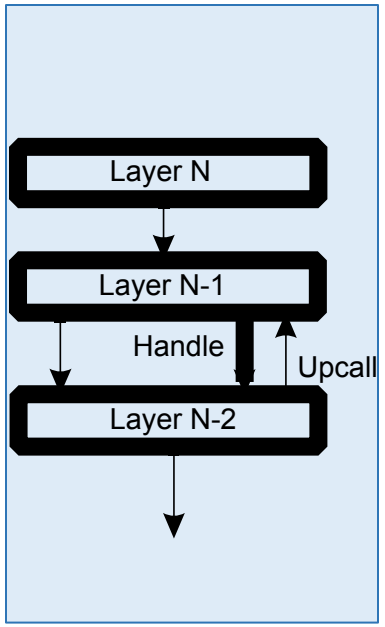
Una aplicación *A*, en el nivel *N*-1, usa una librería L_{os}, en el nivel *N*-3, para comunicarse con el SO; y otra librería especializada L_{math}, en el nivel *N*-2, que también usa L_{os}



Organización estratificada pura



Organización estratificada mezclada



Organización estratificada con *upcalls*

Puede ser necesario que una capa de más abajo llame a su capa inmediatamente superior (*upcall*); p.ej., el SO indica la ocurrencia de un evento, por lo que llama a una operación definida por el usuario a través de una referencia pasada previamente por una aplicación

… y sus beneficios

Separación de intereses (o responsabilidades), de servicios de alto nivel de los de bajo nivel, y de servicios específicos de los generales:

- menor acoplamiento, mayor cohesión, reusabilidad potencial y claridad

La complejidad es encapsulada y es separable

Algunas capas —principalmente las de más “arriba”— pueden ser reemplazadas por nuevas implementaciones

Las capas de más abajo contienen funciones reusables

Algunas capas —principalmente las de más abajo— pueden ser distribuidas

Facilita el desarrollo por equipos

Responsabilidades cohesivas y separación de intereses

Las responsabilidades de los módulos o clases en una misma capa deberían estar fuertemente relacionadas entre sí:

- p.ej., los objetos en la capa UI **deberían** enfocarse en crear ventanas, capturar eventos del mouse y teclado, etc.

  … y **no deberían** contener lógica para calcular impuestos o mover una ficha

- p.ej., los objetos en la capa de la lógica de la aplicación **deberían** enfocarse en calcular el total de una venta o los impuestos, o mover una ficha en un juego de tablero

  … y **no deberían** encargarse de procesar eventos del mouse o del teclado

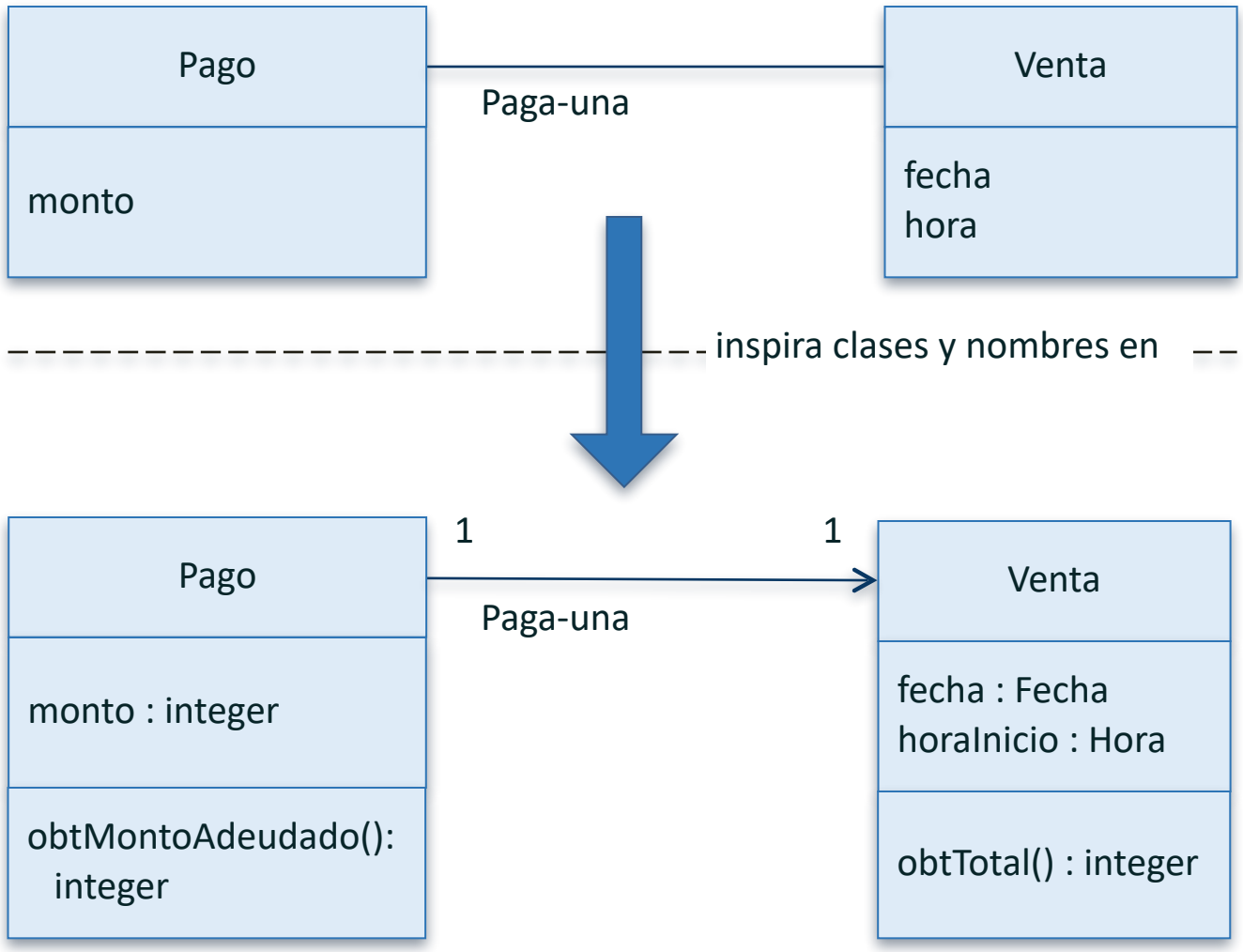
… y no deberían estar mezcladas con responsabilidades de otras capas

Construyamos clases o módulos de software con nombres e información similares a los del dominio en el mundo real

… y asignémoles responsabilidades de la lógica de la aplicación:

- p.ej., en un punto de venta real se producen ventas y pagos
- … → definamos las clases *Venta* y *Pago* y démosles responsabilidades de la lógica de la aplicación, p.ej., un objeto *Venta* es capaz de calcular su total
- estos objetos de software son llamados **objetos del dominio** —representan algo en el espacio del dominio del problema
- … y la capa de la lógica de la aplicación es más precisamente la **capa del dominio** de la arquitectura

Modelo de dominio
(visión de los *stakeholders*)



Capa del dominio de la arquitectura en el Modelo de Diseño

El principio de diseño Separación Modelo-Vista

Los módulos o clases del modelo no deberían tener conocimiento *directo* de los módulos o clases de la vista:

- *modelo* es un sinónimo de la capa de objetos del dominio
- *vista* es un sinónimo de los objetos de la UI
- p.ej., un módulo *Venta* no debería enviar directamente un mensaje a una ventana de la GUI, pidiéndole que despliegue algo, cambie de color, o se cierre

Es clave en el patrón *Modelo-Vista-Controlador* (MVC) [aprox. 1980]:

- el Modelo es la capa Dominio
- la Vista es la capa UI
- los Controladores son los módulos de procesos en la capa Aplicación

Las dos partes del principio Separación Modelo-Vista

No conectar objetos que no son de la UI directamente a objetos de la UI:

- las ventanas están relacionadas con una aplicación particular

  … pero los objetos que no tienen que ver con las ventanas pueden ser reusados en otras aplicaciones o conectados a otras interfaces

… ni poner lógica de la aplicación en los métodos de los objetos de la UI:

- los objetos de la UI sólo deberían inicializar elementos de la UI

  … recibir eventos de la UI

  … y delegar solicitudes relativas a lógica de la aplicación a objetos que no son de la UI (tales como objetos del dominio)

Razones para la separación modelo-vista

Definiciones cohesivas del modelo, focalizadas en los procesos del dominio

Desarrollo independiente de las capas del modelo y de la UI

Reducción del impacto de cambios en los requisitos de la UI sobre la capa del dominio

Fácil conexión de nuevas vistas a la capa del dominio

Posibilidad de múltiples vistas simultáneas sobre el mismo objeto del modelo

Ejecución de la capa del modelo independiente de la capa de la UI

Portabilidad de la capa del modelo a otro *framework* para UI

El patrón Modelo-Vista-Controlador (MVC)

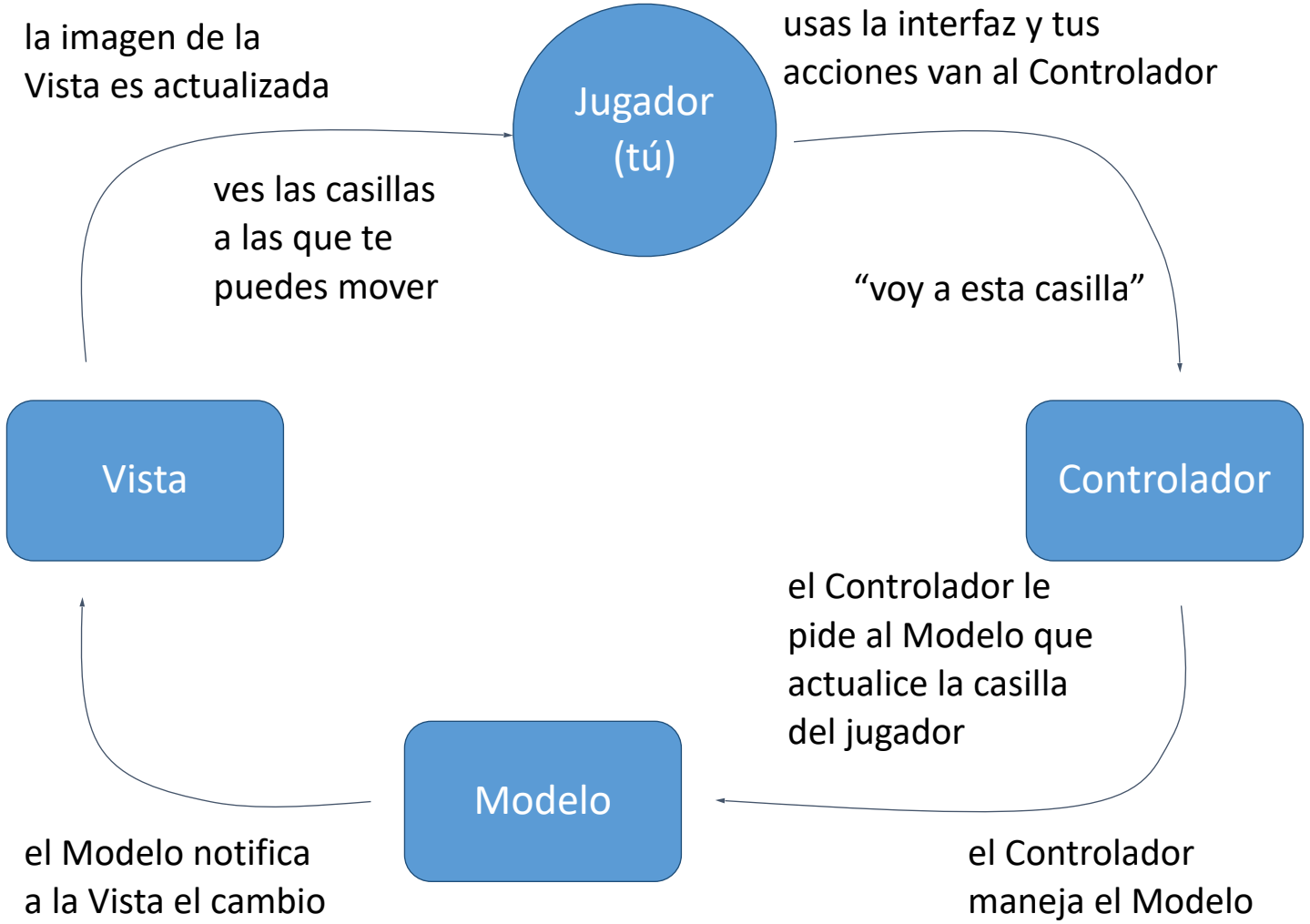
Estás jugando “Trivium”

Usas la interfaz para lanzar el dado, elegir la casilla a la que te quieres mover, y decir si respondiste correctamente o no la pregunta que te hizo el juego

El juego mantiene la información de cada jugador: casilla actual, puntos, etc.

… y controla que se respeten las reglas del juego

La UI es actualizada constantemente con el valor del dado, las casillas a las que te puedes mover, la pregunta que te están haciendo, etc.



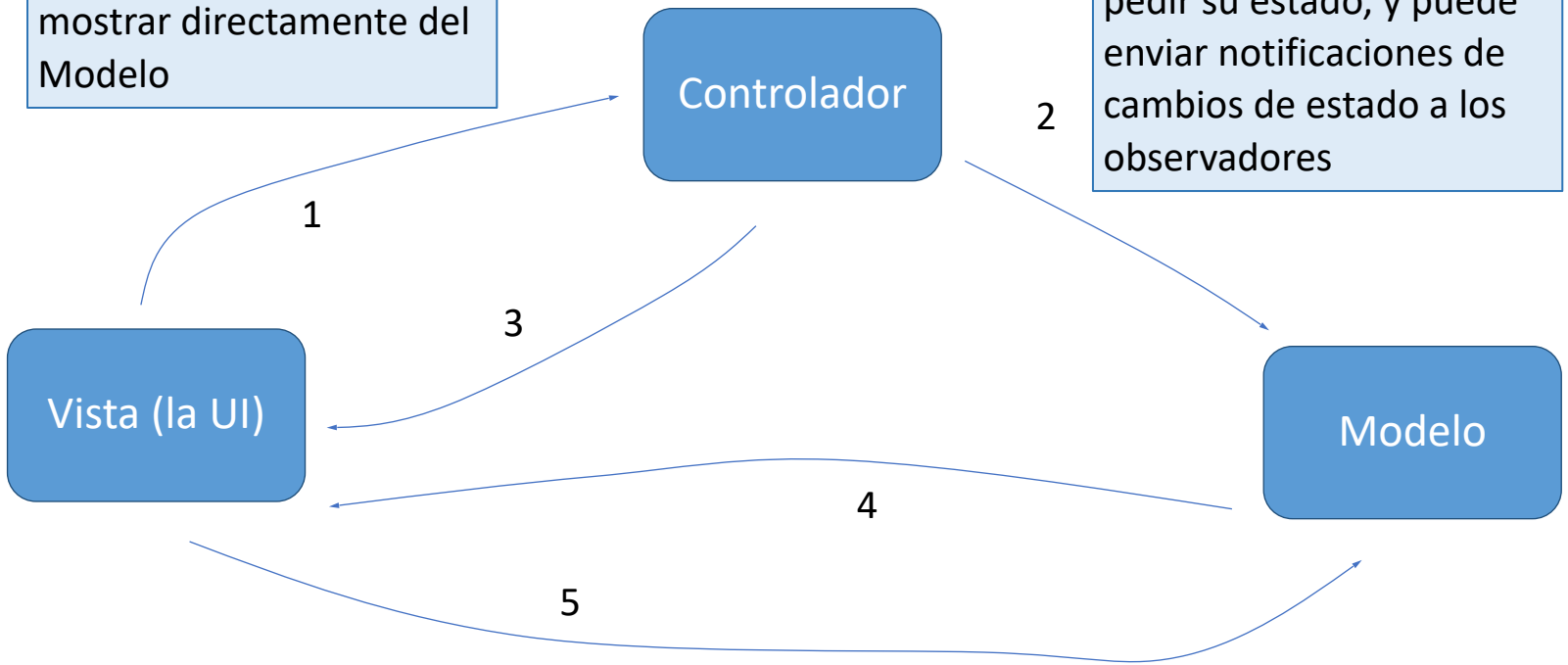
… y un poco más en general

1. Tú —el usuario— hiciste algo—interactúas con la Vista: la Vista se lo cuenta al Controlador
2. El Controlador recibe tus acciones, las interpreta, y le pide al Modelo: “Cambia tu estado”
3. El Controlador también podría tener que pedirle a la Vista: “Cambia tu presentación”
4. El Modelo notifica a la Vista cuando su estado ha cambiado, ya sea por una acción tuya o un cambio interno: “Cambié”
5. La Vista obtiene el estado que tiene que desplegar directamente del Modelo: “Necesito tu información de estado”

Vista (la UI):
Te da una presentación del modelo. Típicamente, obtiene el estado y los datos que necesita mostrar directamente del Modelo

Controlador:
Toma el input del usuario y decide qué significa para el Modelo

Modelo:
Tiene todos los datos, estado y lógica de la aplicación. Ofrece una interfaz para manejar y pedir su estado, y puede enviar notificaciones de cambios de estado a los observadores



MVC es un *patrón compuesto*
(de otros patrones, no es el patrón *Compuesto*)

El Controlador y la Vista son *observadores* del Modelo (patrón **Observador**—diap. 21):

- ambos se registran como observadores del Modelo
- … y son notificados cada vez que el estado del Modelo cambia

La Vista es un objeto configurado con una **Estrategia** proporcionada por el Controlador:

- la Vista delega al Controlador el manejo de las acciones del usuario
- podemos cambiar el comportamiento de la Vista cambiando el Controlador

La Vista es en sí misma un *compuesto* de componentes GUI—aquí sí estamos hablando del patrón **Compuesto** (diap. 22)

- el componente del nivel de más arriba contiene otros componentes, los cuales contienen otros componentes, etc.
- etiquetas, botones, entradas de texto, etc.

Arquitectura cliente-servidor simple

proceso que solicita un servicio de un servidor, mediante el envío de una solicitud y la subsecuenta espera de la respuesta

proceso que implementa un servicio específico; p.ej., un sistema de archivos, una base de datos

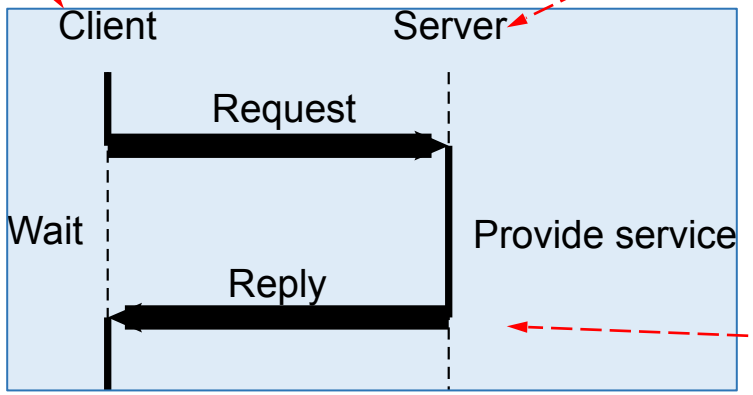


diagrama de secuencia de mensajes que muestra la interacción cliente-servidor

si la red es suficientemente confiable, la comunicación se puede imple-mentar mediante un protocolo simple de tipo solicitud/respuesta

La organización más simple para distribuir físicamente una aplicación cliente-servidor de tres niveles lógicos es tener sólo dos tipos de computadores:

- un computador cliente que contiene sólo los programas que implementan (parte de) el nivel de la interfaz de usuario
- un computador servidor que contiene el resto, es decir, los niveles de procesamiento y de datos

Podemos mover parte de la aplicación al *front end*:

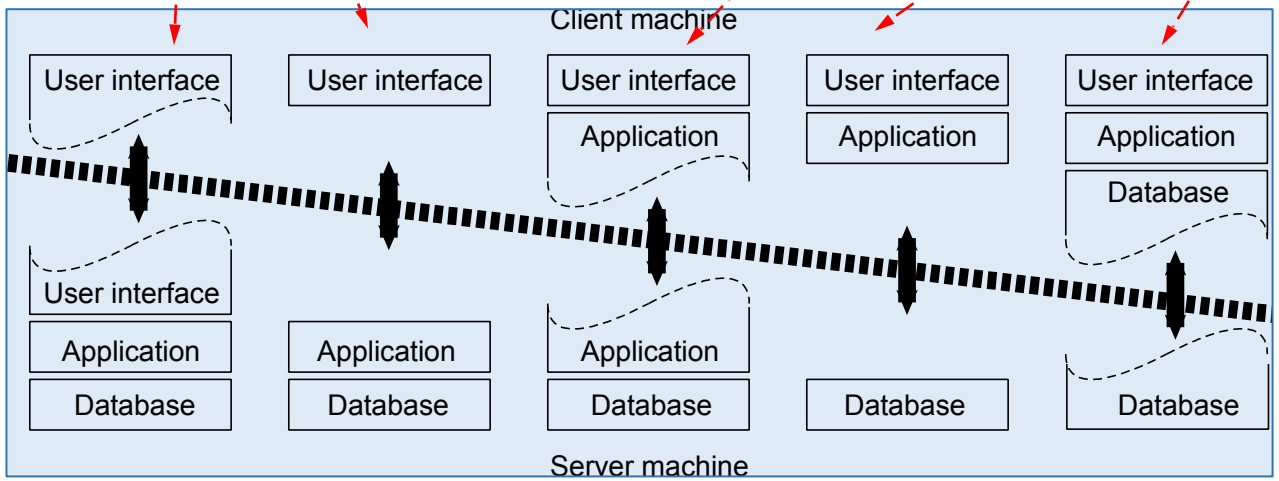
- la aplicación usa un formulario que tiene que ser llenado antes de ser procesado
- un procesador de texto, en que las funciones básicas de edición se ejecutan en el cliente

Típicamente, el cliente es un PC conectado a través de una red a un sistema de archivos o base de datos:

- aplicaciones bancarias
- navegación por la Web, en que el cliente arma una *cache* de páginas visitadas en su disco local

en el cliente, sólo la parte de la UI dependiente del terminal

en el cliente, todo el software de la UI



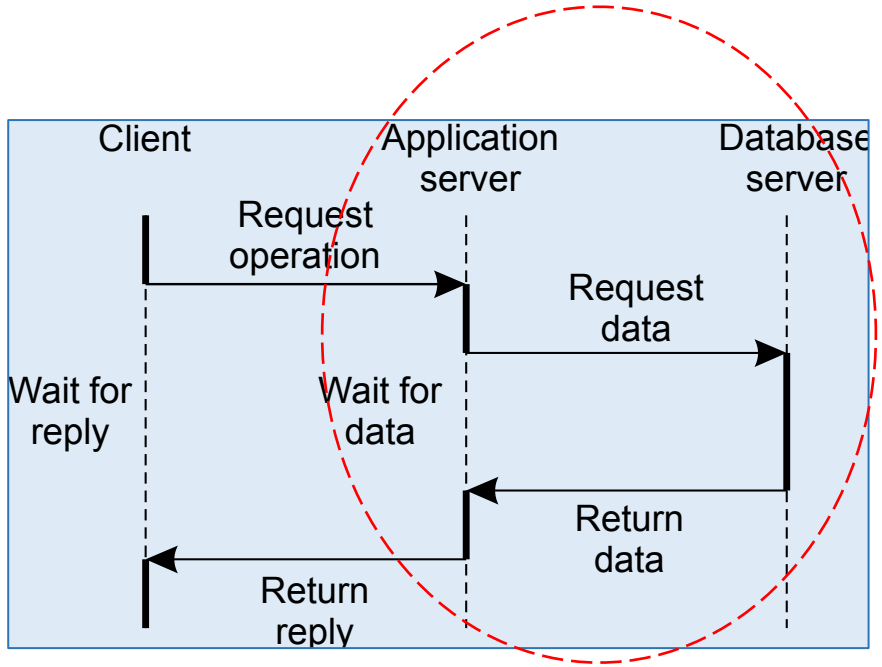
Arquitecturas de dos capas físicas
(2-tiered)

Arquitecturas de tres capas físicas
(3-tiered)

Un servidor a veces tiene que actuar como cliente.

Los programas que forman parte de la capa de procesamiento son ejecutados por un servidor separado ... pero además pueden estar parcialmente distribuidos entre los computadores del cliente y del servidor:

- p.ej., en procesamiento de transacciones, el monitor de transacciones coordina todas las transacciones a través de diferentes servidores de datos



Otro ej. es la organización de los sitios Web:

- un servidor Web actúa como punto de entrada al sitio, pasando las solicitudes a un **servidor de aplicaciones**, en donde ocurre el procesamiento real, y que a su vez interactúa con un **servidor de base de datos**

Todo junto: arquitecturas de tipo software as a service (SaaS)

Siguen el patrón arquitectónico *cliente-servidor* (próx. diap.)

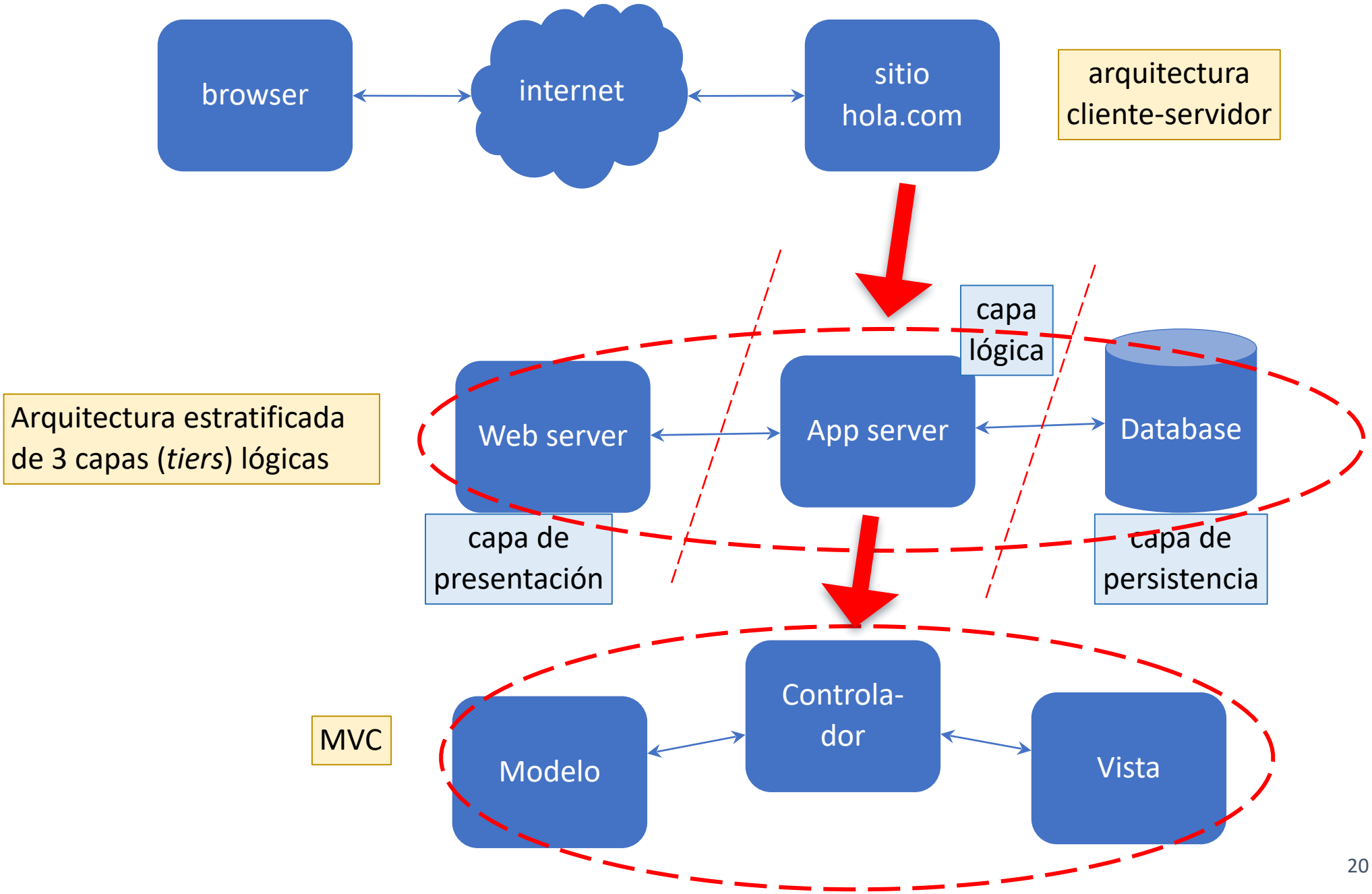
- un **cliente** hace solicitudes y un **servidor** responde a las solicitudes de muchos clientes

El servidor SaaS sigue el patrón *arquitectura de 3 capas* (próx. diap.):

- separa las responsabilidades de los diferentes componentes del servidor

El código de la aplicación vive en la capa de la aplicación y sigue el patrón de diseño MVC:

- el Modelo se preocupa de los recursos de la aplicación
- la Vista presenta información al usuario a través del browser
- el Controlador asocia las acciones del usuario en el browser con el código de la aplicación


browser
internet
sitio
hola.com
arquitectura
cliente-servidor
Arquitectura estratificada
de 3 capas (*tiers*) lógicas
capa
lógica
Web server
App server
Database
capa de
presentación
capa de
persistencia
MVC
Modelo
Controla-
dor
Vista

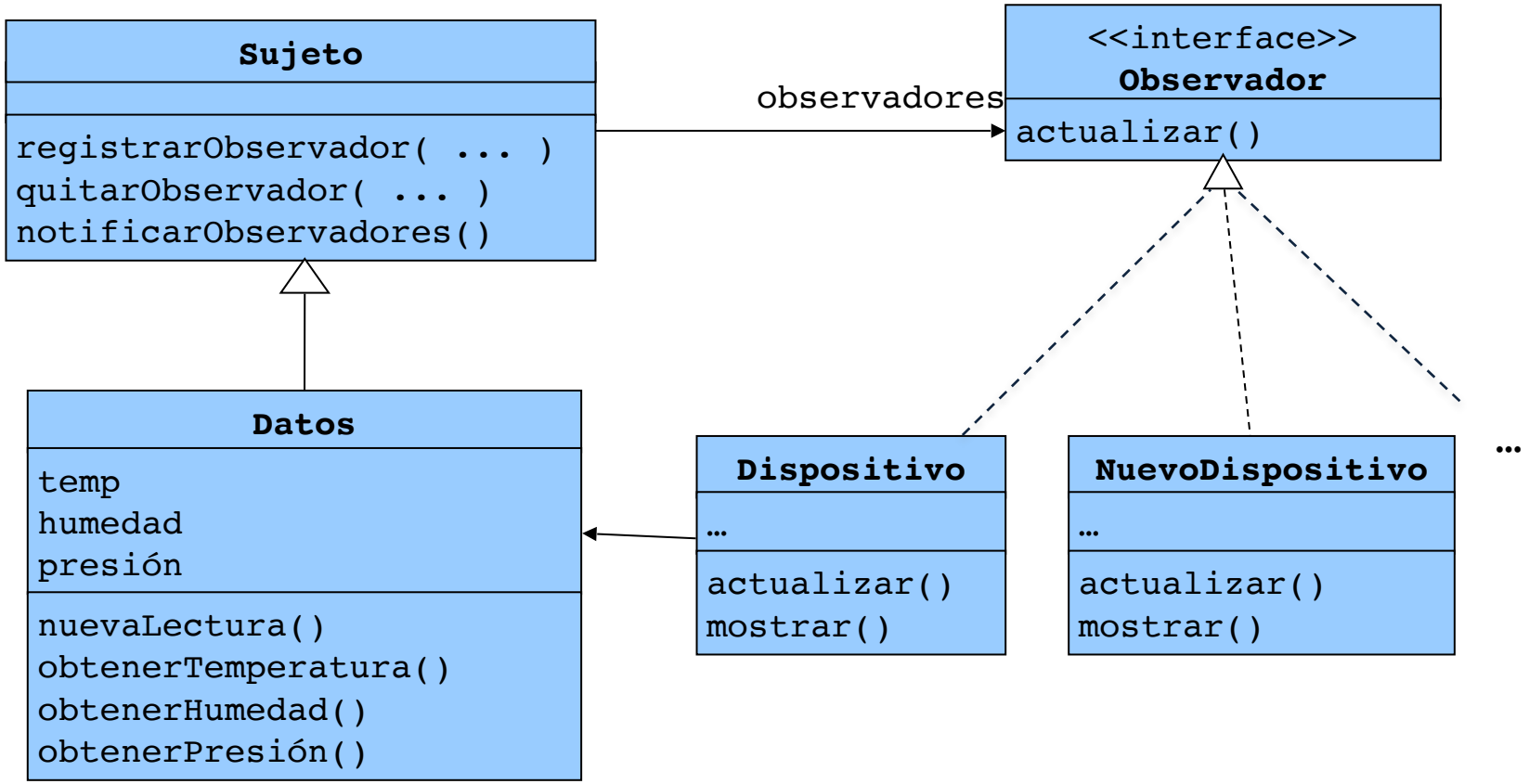
El patrón de diseño ***Observador***

Problema: Diferentes objetos *observadores* están interesados en los cambios producidos en otro objeto *sujeto*;

… el sujeto, o publicador, quiere tener poco, o bajo, acoplamiento con sus suscriptores (los observadores)

Solución: Definir una interfaz "Observador" que sea implementada por los diferentes observadores o suscriptores;

… los observadores pueden subscribirse para ser notificados cuando ocurre un evento



El patrón de diseño ***Compuesto***

Contexto:

- objetos simples —hojas— pueden ser combinados en objetos compuestos
- los clientes tratan el objeto compuesto como simple

Solución:

- definir una interfaz que sea una abstracción de los objetos simples
- un objeto compuesto contiene objetos simples
- tanto las clases simples como las clases compuestas implementan la (misma) interfaz
- al implementar un método de la interfaz, la clase compuesta lo aplica primero a sus objetos simples, y luego combina los resultados