

Introducción a la Arquitectura de Software

Ingeniería de Software – IIC2143

Yadran Eterovic (yadran@ing.puc.cl)

2-2018

Diseño de software, pero a mayor escala

A esta escala más grande, el diseño de un sistema oo típico se basa en varias capas arquitectónicas

P.ej., la capa de la interfaz de usuario

... la capa de la lógica de la aplicación (o del “dominio”)

¿Qué es arquitectura de software?

Es el conjunto de decisiones importantes acerca de la organización de un sistema de software

¿Cuáles decisiones acerca de la organización del sistema?

La selección de los elementos estructurales y sus interfaces, a partir de los cuales el sistema es compuesto

Los comportamientos de estos elementos, especificados en la forma de colaboraciones

La composición de estos elementos, y sus comportamientos, en subsistemas progresivamente más grandes

El estilo arquitectónico que guía esta organización —los elementos, sus interfaces, colaboraciones y composición

¿Qué es la arquitectura lógica de un sistema?

Es la organización a gran escala de las clases del software, p.ej., en

- capas
- subsistemas
- paquetes (en el sentido que le dan algunos lenguajes de programación, p.ej., Java)

¿Qué es la arquitectura física de un sistema?

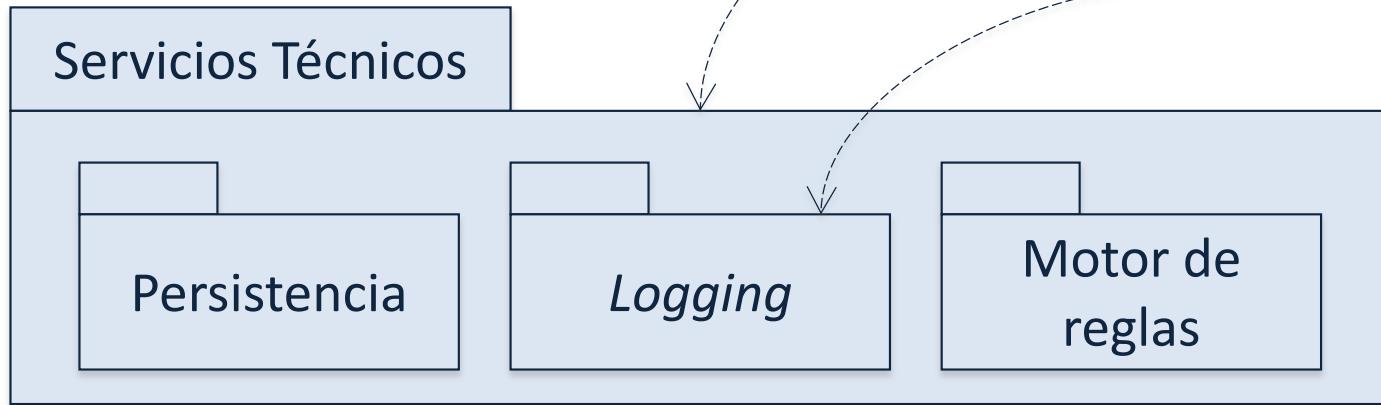
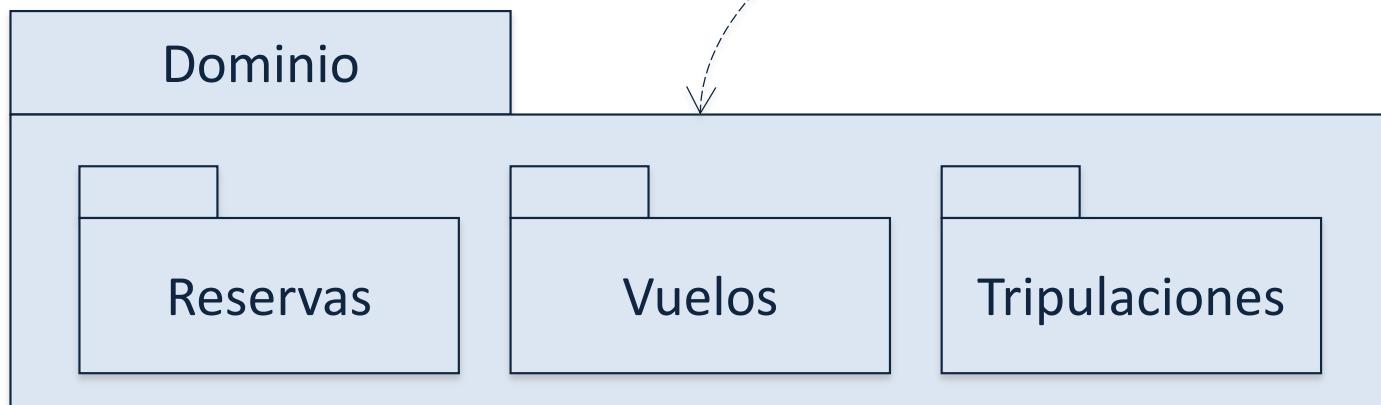
Involucra las decisiones acerca de cómo se distribuirán (o desplegarán) los elementos identificados en la arquitectura lógica

... entre los diversos procesos del sistema operativo o entre los distintos computadores físicos de una red

(más adelante)

¿Qué es una *capa* de software?

Es una agrupación de clases, paquetes, o subsistemas que de manera cohesiva tiene responsabilidad por un aspecto importante del sistema



Capas típicas de un sistema oo

Interfaz de usuario (UI)

Lógica de la aplicación y objetos del dominio:

- objetos de software que representan conceptos del dominio (p.ej., la clase *Venta*) y satisfacen requisitos de la aplicación. tal como calcular el total de una venta

Servicios técnicos:

- objetos y subsistemas de propósito general que proporcionan servicios de apoyo, tales como comunicación con una base de datos o registro de errores, normalmente independientes de la aplicación y reusables por varios sistemas

El ejemplo en la diapositiva anterior está ilustrado usando el *diagrama de paquetes* de UML

¿Cómo organizamos un sistema en capas?

Las capas son organizadas de la siguiente manera:

- las de más “arriba” (p.ej., la capa de UI) son más específicas a la aplicación y llaman a los servicios de las de más “abajo” (y no vice versa —pauta de diseño)
- las de más “abajo” (p.ej., la capa de Servicios Técnicos) son servicios generales y de bajo nivel

En una arquitectura por capas (o estratificada) estricta —p.ej., el *stack* de protocolos de una red— una capa llama sólo a los servicios de la capa directamente bajo ella

... no así en una arquitectura relajada, común en sistemas de información (e ilustrada en el ejemplo anterior)

(Un paréntesis: Paquetes en UML ...

Un *paquete* es el elemento para modelar agrupación de elementos:

- es contenedor, y dueño, de otros elementos del modelo
- define su propio espacio de nombres, dentro del cual todos los nombres deben ser únicos

¿Qué tipos de elementos puede agrupar un paquete en UML?

Básicamente, cualquier cosa:

- clases, otros paquetes, casos de uso, etc.
- la anidación de paquetes es muy común
- p.ej., en el ejemplo, el paquete *Dominio* agrupa los paquetes *Reservas*, *Vuelos* y *Tripulaciones*

Un paquete es un espacio de nombres:

- p.ej., una clase *Vuelo* puede estar definida en dos paquetes
- para distinguir elementos con el mismo nombre en paquetes diferentes, pueden usarse nombres “totalmente calificados”, p.ej., *Dominio::Reservas*

Los diagramas de paquetes ilustran la arquitectura lógica de un sistema

El ícono para un paquete es una carpeta:

- el nombre puede ir en la etiqueta (si se muestra el contenido) o en el cuerpo de la carpeta

Una capa arquitectónica puede ser modelada como un paquete:

- p.ej., la capa de interfaz de usuario modelada como un paquete llamado UI

Las líneas de dependencia del UML (----->) se usan para mostrar dependencias entre paquetes:

- p.ej., el paquete *UI* depende del paquete *Dominio*

Hay cinco tipos comunes de dependencias entre paquetes

«**uses**» : un elemento del paquete cliente usa un elemento público del paquete proveedor

«**import**» : elementos públicos del espacio de nombres del proveedor son agregados como elementos públicos al espacio de nombres del cliente

«**access**» : elementos públicos del espacio de nombres del proveedor son agregados como elementos privados al espacio de nombres del cliente

«**trace**» : evolución histórica de un elemento en otra versión más desarrollada

«**merge**» : elementos públicos del paquete proveedor son mezclados con los elementos del paquete cliente

... dos ejemplos de diagramas de paquetes)

Ambos tomados de un documento de arquitectura de software real

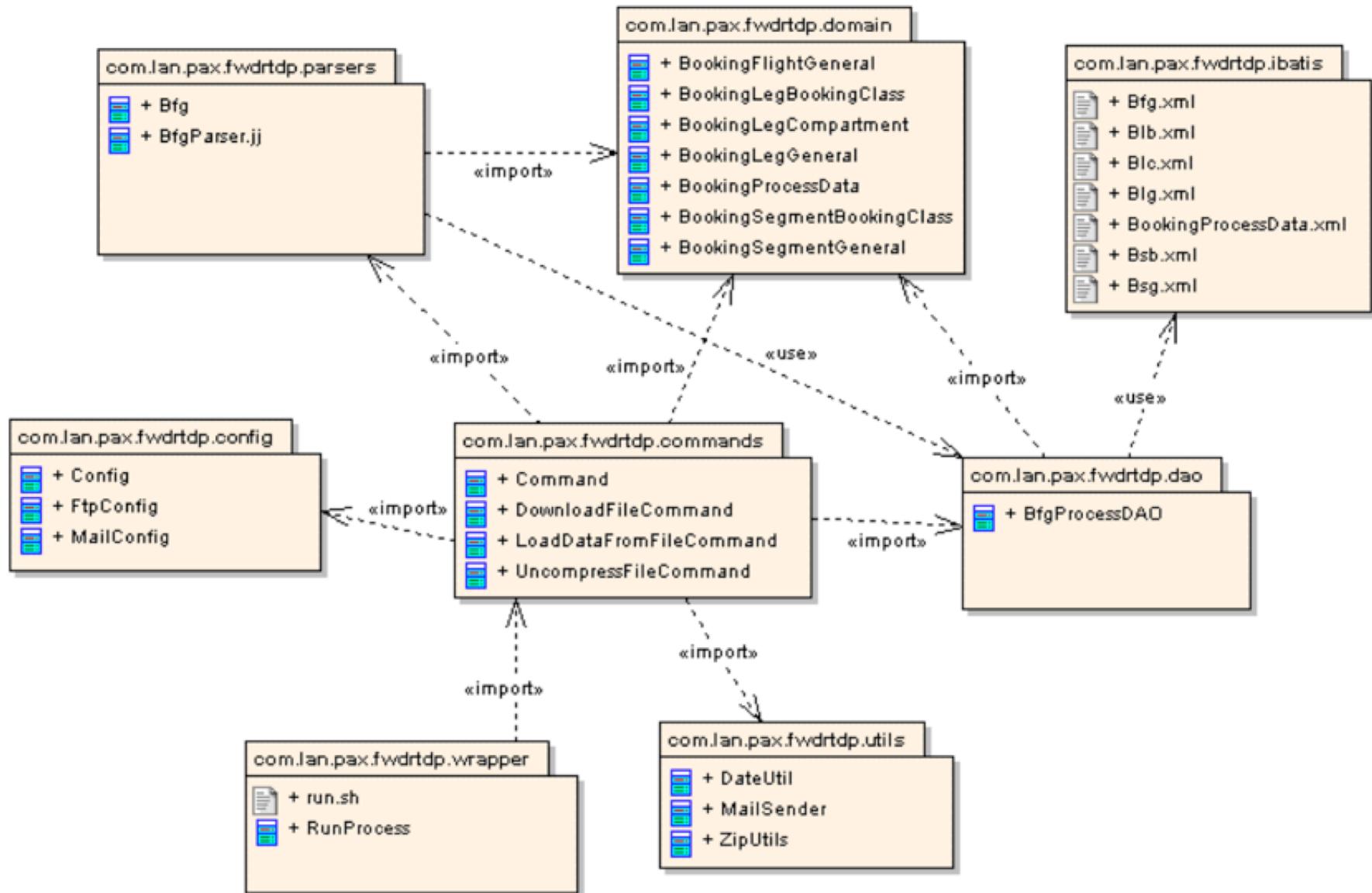
El primero (dentro del marco “class packages”) ilustra ocho paquetes,

... la mayoría de los cuales agrupa clases públicas (+), y las dependencias entre ellos, de tipos «*uses*» e «*import*»

El segundo (dentro del marco “class fwdrtdp.ejb”) ilustra dos paquetes principales,

... cada uno de los cuales contiene otros paquetes, y las dependencias entre ellos, también de tipos «*uses*» e «*import*»

class packages



```
class fwdrtdp-ejb
```

fwdrtdp-ejb

com.lan.pax.fwdrtdp.persistence.entities

+ Flight
+java.io.Serializable

com.lan.pax.fwdrtdp.persistence.dao

+ FlightDAOImpl
+ GenericDAOImpl
IBatisUtil
+ FlightDAO
+ GenericDAO

com.lan.pax.fwdrtdp.session

+ FlightBean
+javax.ejb.SessionBean

com.lan.pax.fwdrtdp.business.services

+ FlightService

fwdrtdp-ejbClient

com.lan.pax.fwdrtdp.dto

+ FlightDTO
+java.io.Serializable

com.lan.pax.fwdrtdp.business.exceptions

+ BusinessException
+ InternalErrorException
+ java.lang.Exception
+ java.lang.RuntimeException

«import»

«import»

«use»

«import»

«use»

«import»

«import»

«use»

«use»

Usar capas ayuda a enfrentar problemas como los siguientes

Los cambios en el código fuente se replican por todo el sistema:

- muchas partes del sistema están muy acopladas

La lógica de la aplicación está entrelazada con la interfaz de usuario:

- no puede ser reusada con una interfaz diferente, o distribuida a otro nodo de procesamiento

Los servicios técnicos o la lógica del negocio están entrelazados con la lógica específica a la aplicación:

- no pueden ser reusados, o distribuidos a otro nodo, o reemplazados fácilmente por otras implementaciones

El acoplamiento es alto entre diferentes áreas temáticas:

- es difícil dividir el trabajo de manera clara entre diferentes desarrolladores

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/Infra-estructura Técnicos de Bajo Nivel):

- mayor rango de aplicabilidad

... 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 Dominio y Servicios Técnicos— pueden ser distribuidas

Facilita el desarrollo por equipos

Responsabilidades cohesivas y separación de intereses

Las responsabilidades de los objetos en una capa deberían estar fuertemente relacionadas entre sí:

- 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
 - 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

Diseñando la lógica de la aplicación con objetos

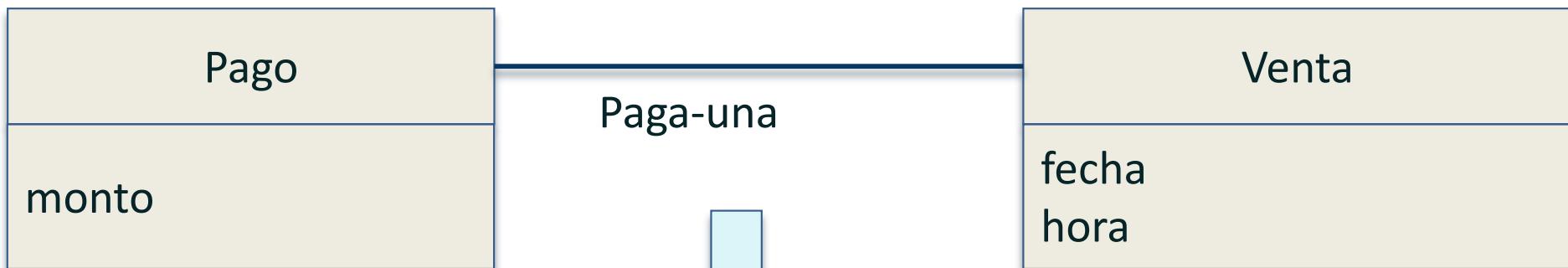
Construyamos objetos de software con nombres e información similares a los del dominio en el mundo real

... y asignémosles responsabilidades de la lógica de la aplicación:

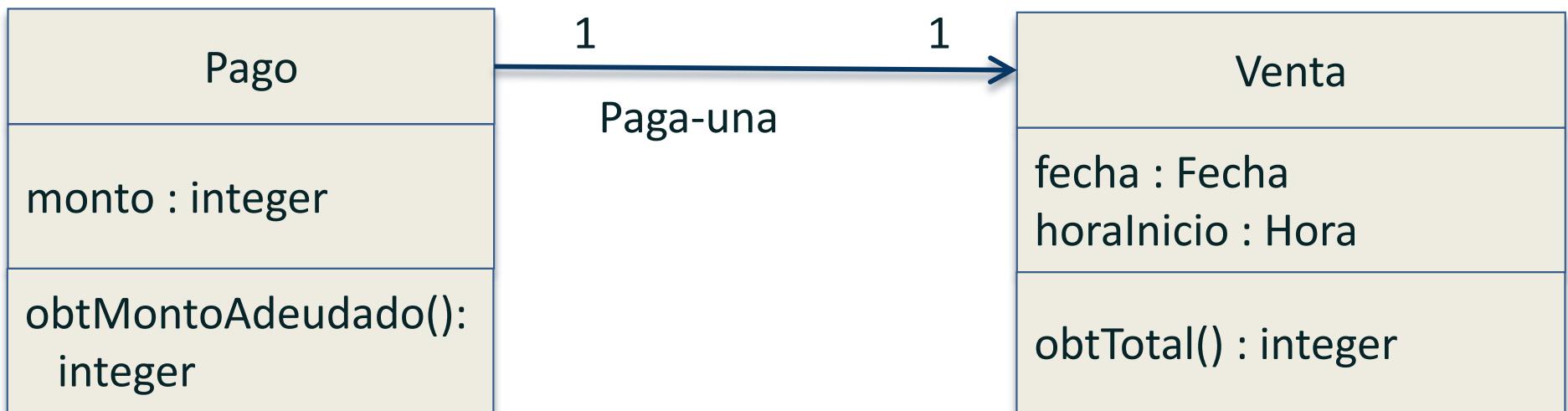
- p.ej., en un punto de venta real se producen ventas y pagos
- ... luego, definamos las clases *Venta* y *Pago* y démosles responsabilidades de la lógica de la aplicación, p.ej., un objeto *Venta* es capaz de calcular su total
- estos objetos de software son llamados **objetos del dominio** —representan 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*)



inspira objetos y nombres en



Capa del dominio de la arquitectura en el Modelo de Diseño

El principio de separación *modelo-vista*

Los objetos del modelo no deberían tener conocimiento *directo* de los objetos 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 objeto *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 objetos de procesos en la capa Aplicación

El principio *separación modelo-vista* tiene dos partes

No conectemos 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 pongamos lógica de la aplicación en los métodos de los objetos de la UI:

- los objetos de la UI solo 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)

Varias 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”

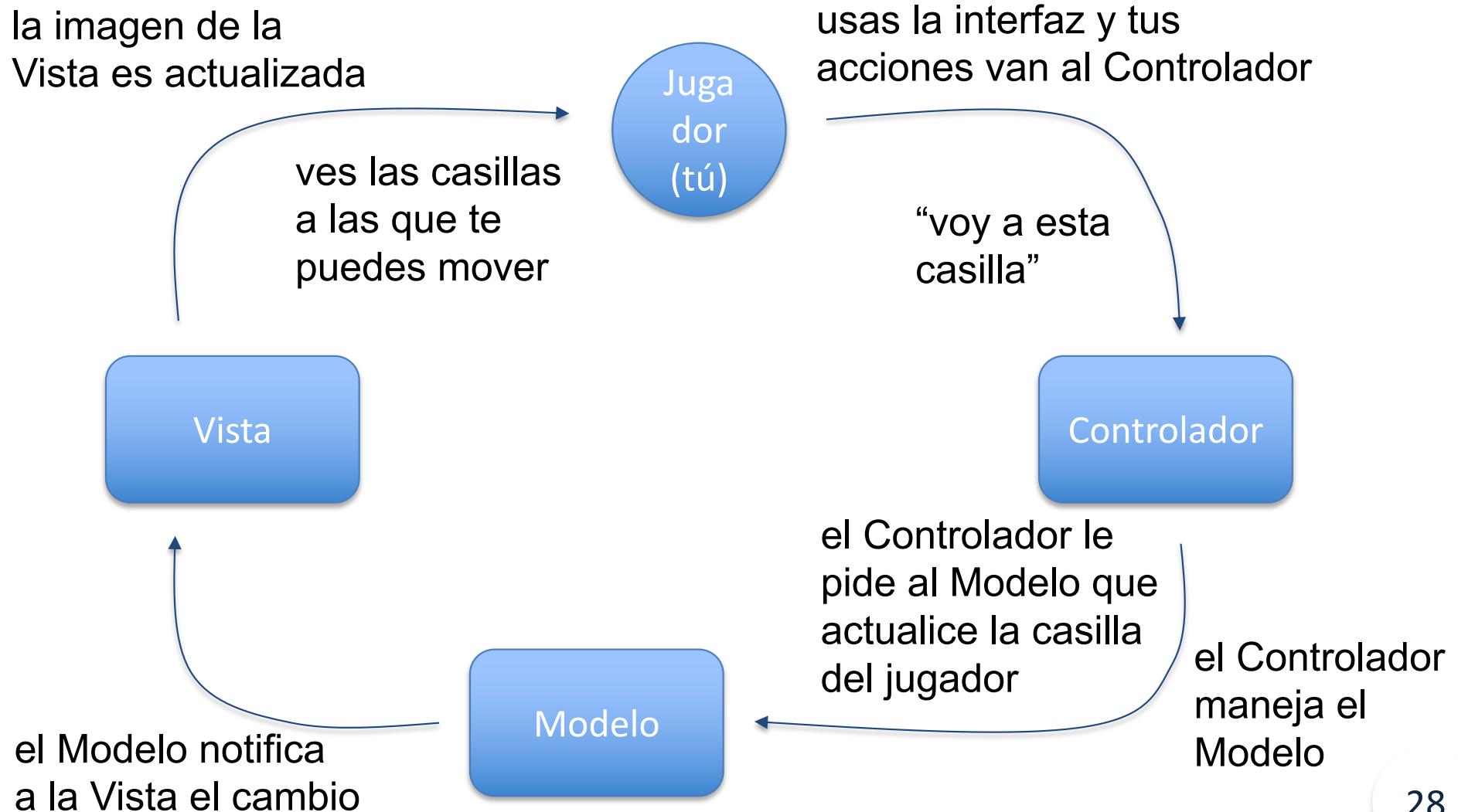
Usas la interfaz para lanzar el dado, elegir la casilla a la que te quieres mover, y decir si respondiste correctamente o no la pregunta que te hizo el juego

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.

El patrón MVC gráficamente



... y un poco más en detalle

Vista (la UI):

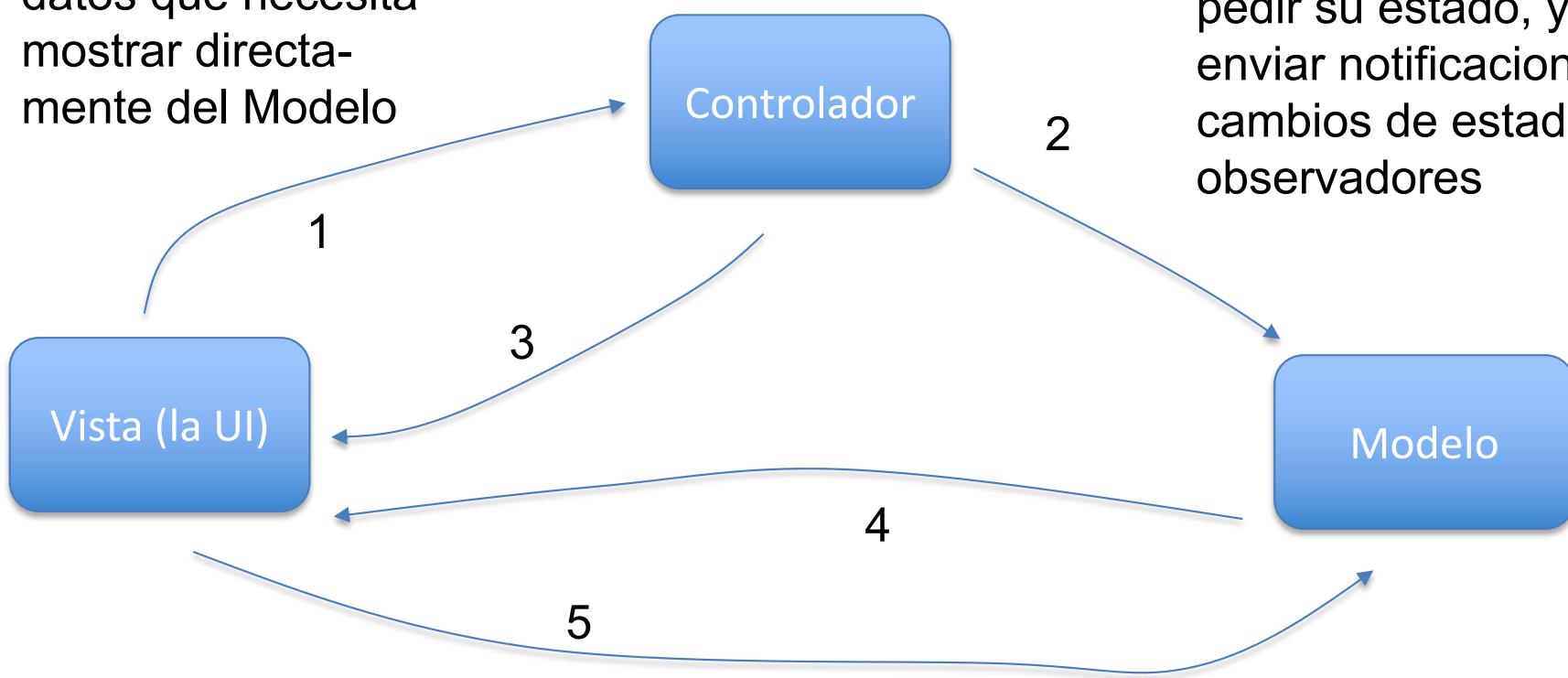
Te da una presentación del modelo. Típicamente, obtiene el estado y los datos que necesita mostrar directamente del Modelo

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



... las cinco acciones

- 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”

MVC es un *patrón compuesto* (no el patrón *Compuesto*)

El Controlador y la Vista son *observadores* del Modelo:

- 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* —el Controlador proporciona la *estrategia*:

- 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*:

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

Análisis arquitectónico

Especialización del análisis de requisitos, enfocado en requisitos que influyen fuertemente en la arquitectura:

- p.ej., la necesidad de un sistema muy seguro

Consiste en identificar factores que deberían influenciar la arquitectura, entender sus variabilidades y prioridades, y resolverlos

Es difícil por varias razones

Hay que saber qué preguntas hacer

Hay que sopesar las concesiones mutuas

Hay que conocer las diversas formas de resolver un factor arquitectónicamente importante:

- desde omitirlo
- ... hasta diseños elaborados
- ... y hasta productos de terceros

... pero también es importante por varias razones

Reduce el riesgo de pasar por alto algo importante en el diseño del sistema

Evita hacer esfuerzos excesivos en problemas de baja prioridad

Ayuda a alinear el producto con los objetivos del negocio

Puntos de cambio en un sistema de software

Punto de variación:

- variaciones en el sistema o en los requisitos vigentes
- p.ej., múltiples interfaces para subsistemas calculadores de impuestos

Punto de evolución:

- puntos de variación posibles que podrían presentarse en el futuro, pero que no están presentes actualmente

Son temas claves recurrentes en análisis arquitectónico

Así, al análisis arquitectónico le concierne ...

- ... la identificación y resolución de los requisitos no funcionales del sistema (p.ej., seguridad)
- ... en el contexto de los requisitos funcionales (p.ej., al procesar una venta)
- ... incluyendo la identificación de puntos de variación
- ... y los más probables puntos de evolución

Ejemplos de problemas que deben ser identificados y resueltos al nivel de arquitectura

¿Cómo afectan el diseño

... los requisitos de confiabilidad y tolerancia a fallas?

... los requisitos de adaptabilidad y configurabilidad?

¿Cómo afectan a la rentabilidad los costos de licenciamiento de algunos subcomponentes?

¿Cómo afecta a la arquitectura el empleo de nombres de marca?

Dos pasos del análisis arquitectónico: El primero

1. (la ciencia) Identificar y analizar los **requisitos no funcionales** que tienen un impacto en la arquitectura (aunque los requisitos funcionales también podrían influir) —**factores arquitectónicos**:

- podría considerarse parte del análisis de requisitos regular

... y el segundo

2. (el arte) Analizar alternativas y crear soluciones que resuelvan el impacto —**decisiones arquitectónicas**:

- desde “eliminar el requisito”, hasta una solución a la medida, hasta “detener el proyecto”, hasta “contratar un experto”

Factores arquitectónicos

Cualquiera de los requisitos FURPS+ puede influir en la arquitectura:

- p.ej., poco plazo y pocas personas, pero suficiente dinero, favorece comprar o contratar especialistas, en lugar de desarrollar internamente

FURPS+

FURPS:

- funcional
- usabilidad
- confiabilidad (*reliability*)
- desempeño (*performance*)
- soportabilidad

+:

- implementación (recursos, lenguajes y herramientas, hardware, ...), interfaces (con otros sistemas), operaciones (gestión operacional del sistema), empaquetamiento, legal (licenciamiento)

Factores arquitectónicos más influyentes

Son los de *funcionalidad, confiabilidad, desempeño, soportabilidad, implementación, e interfaz*:

- los **atributos de calidad no funcionales** le dan a una arquitectura particular su “sabor” único
- p.ej., en el sistema PdV, el diseño para permitir diversas componentes de terceros con interfaces únicas,
- ... y el diseño para permitir conectar fácilmente distintos conjuntos de reglas de negocio

Factores como restricciones

Hay factores técnicos y organizacionales que aparecen como restricciones:

- p.ej., debe correr en Linux
- p.ej., el presupuesto para comprar componentes es X

Escenarios de calidad

No es muy útil establecer que “el sistema será fácil de modificar” sin alguna medida de lo que significa

Típicamente, metas de desempeño y tiempo medio entre fallas aparecen cuantificados

Escenarios de calidad (ahora sí)

... pero conviene registrar todos (o al menos, la mayoría de) los factores como afirmaciones medibles —**escenarios de calidad**:

- p.ej., cuando la venta es enviada al sistema remoto de cálculo de impuestos, el resultado vuelve dentro de 2 segundos la “mayoría” de las veces, medido bajo condiciones de carga “promedio”

“Mayoría” y “promedio” necesitan definiciones adicionales:

- un escenario de calidad no es válido hasta que esté totalmente especificado y sea verificable (*testable*)

Aplicaciones de tipo *software as a service* (SaaS)

Siguen el patrón *cliente-servidor*:

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

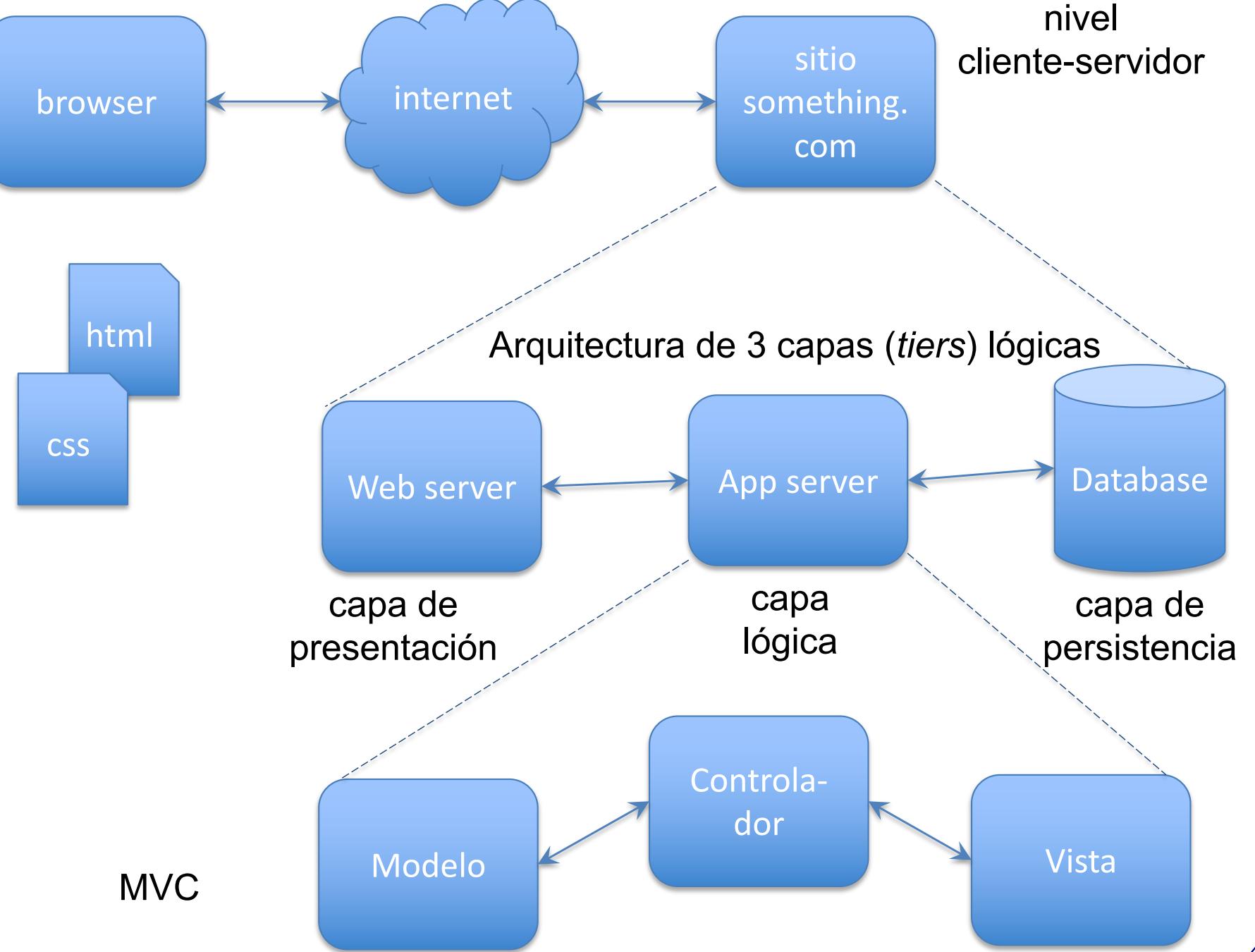
El servidor SaaS sigue el patrón *arquitectura de 3 capas* (o *tiers*):

- separa las responsabilidades de los diferentes componentes del servidor

El código de la aplicación SaaS 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



SaaS: Sistema cliente-servidor que usa los estándares abiertos de la www

Separar los clientes de los servidores permite que cada tipo de programa sea muy especializado según su tarea, aunque ambos son comparablemente complejos:

- el cliente puede tener una UI *responsive* y atractiva y se especializa en interactuar con el usuario y enviar solicitudes al servidor en nombre del usuario
- el servidor se concentra en servir eficientemente a muchos clientes simultáneamente

Browsers y servidores

Los *browsers* son clientes usados por millones de personas:

- “cliente universal”

El servidor del sitio Web puede ser instalado en cientos de computadores y servir eficientemente muchas copias del mismo sitio a millones de usuarios

Arquitectura alternativa: *peer-to-peer*

La usan algunos servicios basados en Internet, p.ej., BitTorrent (protocolo de compartimiento de archivos)

Cada participante es tanto un cliente como un servidor —cualquier participante puede pedirle información a cualquier otro participante

Un mismo programa debe comportarse como cliente y también como servidor:

- puede argumentarse que es una arquitectura más flexible
- ... pero es más difícil especializar el programa para que haga realmente bien cualquiera de esas dos funciones

Comunicación a nivel cliente-servidor

El browser y el servidor se comunican usando el HyperText Transfer Protocol (HTTP), que a su vez usa TCP/IP para intercambiar secuencias ordenadas de bytes confiablemente

Cada computador conectado a una red TCP/IP tiene una dirección IP, tal como 128.32.244.172; el Domain Name System (DNS) permite usar nombres fáciles de recordar, tal como `www.ing.uc.cl`

Comunicación a nivel cliente-servidor

Cada aplicación que corre en un computador particular debe “escuchar” en un puerto TCP específico, numerados de 1 a 65,535; los servidores (Web) HTTP usan el puerto 80

Un Uniform Resource Identifier (URI) le da un nombre a un recurso disponible en Internet, del tipo http:...

HTTP es *stateless*

... ya que toda solicitud es independiente y no tiene relación con ninguna de las solicitudes anteriores

Una aplicación que sabe donde uno está tiene que tener su propio mecanismo para hacerlo:

- nada acerca de una solicitud HTTP recuerda esta información

Cookies

Las *cookies* de HTTP asocian un browser particular de un usuario con información en el servidor correspondiente a esa sesión del usuario

... pero es responsabilidad del browser —no de HTTP ni de la aplicación— asegurarse de que las cookies correctas sean incluidas en cada solicitud HTTP:

- los protocolos stateless simplifican el diseño del servidor
- ... a costa del diseño de la aplicación

Representación de la información que intercambian cliente y servidor: HTML ...

Un documento HTML (*HyperText Markup Language*) es una colección de elementos anidados jerárquicamente:

- cada uno empieza con un tag escrito entre < y > que puede tener atributos

Un selector es una expresión que identifica un elemento HTML en un documento usando una combinación del nombre del elemento, de su id (atributo que debe ser único en una página), y de su clase

... y CSS

CSS (*Cascading Style Sheets*) es un lenguaje de *hojas de estilo* para describir atributos visuales de los elementos en una página Web:

- una hoja de estilo asocia propiedades visuales con selectores

La opción “Developer Tools” del browser permiten examinar tanto la estructura de una página como su hoja de estilo

La arquitectura de 3 capas (o *tiers*) lógicas

Capa de presentación:

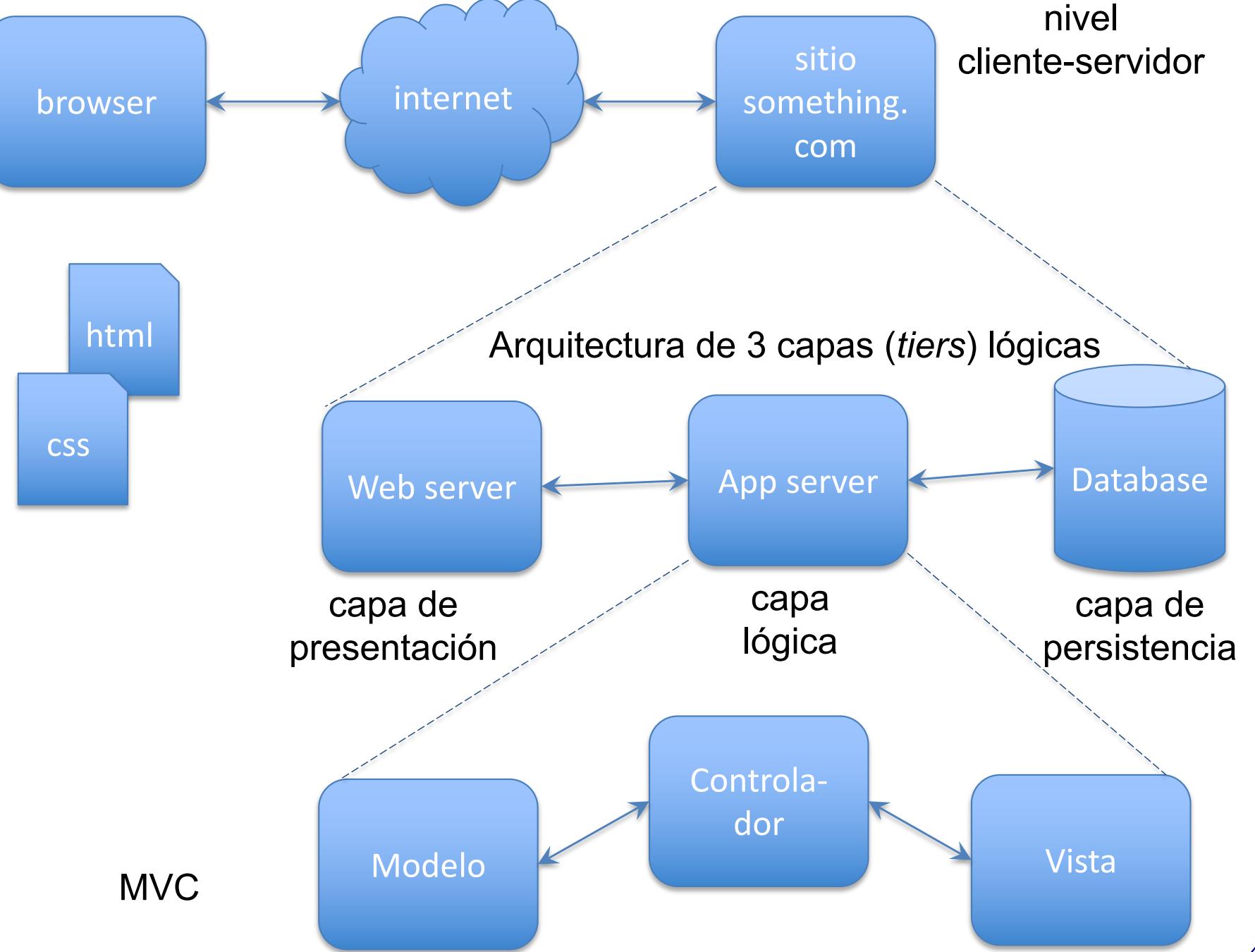
- servidor HTTP, o simplemente “servidor Web”, que produce vistas y acepta solicitudes del mundo exterior (los usuarios) y las reenvía a la capa lógica

Capa lógica:

- donde corre la aplicación SaaS propiamente tal, soportada por un servidor de aplicaciones cuya función es esconder la mecánica de bajo nivel del HTTP del programador

Capa de persistencia:

- necesaria porque HTTP es *stateless*, pero la aplicación maneja información que debe permanecer almacenada a lo largo de varias solicitudes HTTP (p.ej., datos de la sesión, información de login y profile del usuario)
- típicamente, bases de datos (p.ej., MySQL, PostgreSQL)



Las capas son lógicas

En un sitio con poco contenido y poco tráfico, el software de las tres capas podría correr en un solo computador físico

... y cada una puede ocupar varios computadores

En otros casos es más común que cada capa ocupe uno o más computadores físicos:

- las solicitudes HTTP que llegan son enviadas a uno de varios servidores Web
- ... el cual a su vez selecciona uno de varios servidores de aplicaciones para manejar la generación de contenido
- ... permitiendo que los computadores sean agregados o quitados de cada capa como sea necesario para manejar la demanda

El único problema es la capa de persistencia

La arquitectura Modelo-Vista-Controlador: Los modelos

Una aplicación organizada según MVC, como sabemos, consiste en tres tipos principales de código:

- modelos, vistas, y controladores

Los **modelos** se preocupan de los datos manejados por la aplicación:

- cómo almacenarlos, cómo ejecutar operaciones sobre ellos, y cómo cambiarlos
- la aplicación tiene un modelo para cada tipo de entidad que maneja; p.ej., cursos, y mallas curriculares
- contienen el código que se comunica con la capa de almacenamiento

La arquitectura Modelo-Vista-Controlador: Las vistas

Las **vistas** son presentadas al usuario

... y contienen información sobre los modelos con los cuales los usuarios pueden interactuar:

- son la interfaz entre los usuarios del sistema y sus datos
- cada modelo de una aplicación puede estar asociado con varias vistas
- p.ej., listar todos los cursos del quinto semestre de mi malla curricular
 - ... listar todos los cursos ofrecidos por el DCC este semestre
 - ... mostrar los detalles de un curso
 - ... inscribirme en un curso

La arquitectura Modelo-Vista-Controlador: Los controladores

Los **controladores** median la interacción en ambas direcciones:

- cuando el usuario interactúa con una vista (haciendo click sobre algo en una página Web), se invoca una acción específica del controlador correspondiente a esa actividad del usuario
- cada controlador corresponde a un modelo
- cada acción del controlador es manejada por un método dentro del controlador
- el controlador le puede pedir al modelo que vaya a buscar o modifique información
- ... y luego decide qué vista le presentará a continuación al usuario como respuesta a la acción inicial de éste

La estructura de una aplicación SaaS

1. Un cliente Web (un browser) solicita la página de una aplicación a un servidor Web
2. El servidor obtiene contenido de la aplicación y se lo envía al browser
3. El browser muestra el contenido y cierra la conexión HTTP

La estructura de una aplicación SaaS

1. Un cliente Web (un browser) solicita la página de una aplicación a un servidor Web:

- el browser construye una solicitud HTTP usando un URI de la forma `http://...` para contactar al servidor HTTP
- el servidor recibe la solicitud por un recurso, p.ej., (la lista de todos los) “cursos”

La estructura de una aplicación SaaS

2. El servidor obtiene contenido desde la aplicación y se lo envía al browser:

- el servidor Web llama a código de la aplicación que está en la capa de la aplicación; este código genera el contenido de la página usando información de cursos almacenada en la capa de persistencia
- el servidor Web devuelve contenido en HTML usando HTTP
- ... el HTML puede contener referencias a imágenes u otra información para ser incluida en la página desplegada
- ... y también referencias a una hoja CSS que describe los atributos visuales de la página

La estructura de una aplicación SaaS

3. El browser muestra el contenido y cierra la conexión HTTP:

- el browser trae la información referenciada repitiendo los cuatro pasos anteriores
- el browser despliega la página según las directrices de formato de la CSS e incluyendo cualquier elemento referenciado