Laboratorio de Ingeniería del Software

Diseño estrateǵico y microservicios

Contenidos

- Diseño estratećico
  - Contextos delimitados
- Arquitectura dirigida por eventos
- Microservicios

Diseño estratégico

Diseño estratégico

- Cuando los sistemas crecen, es difícil seguir su diseño al nivel de objetos individuales
  - Necesitamos técnicas para manipular y abarcar modelos grandes
- En un sistema grande hay que encontrar el punto medio entre:
  - Modularidad
    - Que nos permite comprender algo grande partiéndolo en partes más pequeñas
  - Integración
    - Que nos permite que esas partes funcionen bien conjuntamente
- Los principios de diseño estratégico ayudarán a que nuestros modelos de dominio tengan menos interdependencias entre sus partes y sean más fáciles de entender y analizar

La integridad del modelo

- Puede que el requisito más importante de cualquier modelo sea la consistencia interna
  - Que cada término en cada lugar que se use signifique siempre lo mismo
  - Que no tenga contradicciones
- Sin esta consistencia lógica, un modelo no tiene sentido
- En sistemas grandes que integran muchos otros (por ejemplo un sistema empresarial que integra los sistemas de cada departamento), nos encontraremos muchos modelos distintos trabajando juntos
  - Habrá que establecer mecanismos para determinar qué partes del sistema pueden diverger y qué partes tienen que estar unificadas

La integridad del modelo

- Tener un modelo de todo que esté unificado sería lo ideal
  - En un sistema grande generalmente esto no será factible, o será prohibitivo en términos económicos
- A pesar de ello, hay quien lo intenta. Pero hay muchos riesgos:
  - Puede que haya que reemplazar varios sistemas legados al mismo tiempo
  - El proyecto puede ser tan grande que la coordinación sea un enorme problema
  - Puede haber sistemas subcontratados y que no podemos cambiar
  - Partes especializadas del sistema pueden verse forzadas a adaptarse a un modelo genérico, que sirve más o menos a todos pero que no es el que necesitarían realmente
    - Hacer un modelo que satisfaga a todos, generalmente conduce a opciones complejas que lo hacen más difícil de usar (un ejemplo del "diseño por comité")
- Que haya distintos modelos puede ser causa de diferentes prioridades de la gerencia de la empresa, o de su organización interna y sus procesos de desarrollo
  - Incluso si no hubiera barrera técnicas y de diseño, podría no ser factible unificar

Un camello es un caballo diseñado por un comité

La integridad del modelo

- Cuando un modelo unificado no resulta práctico debemos hacernos una imagen clara de la situación y tomar algunas medidas
  - Asegurar que ciertas partes críticas están unificadas
  - Asegurar que las partes no unificadas no causen problemas graves
- Para esto tenemos que identificar las fronteras entre los distintos modelos, y las relaciones entre ellos

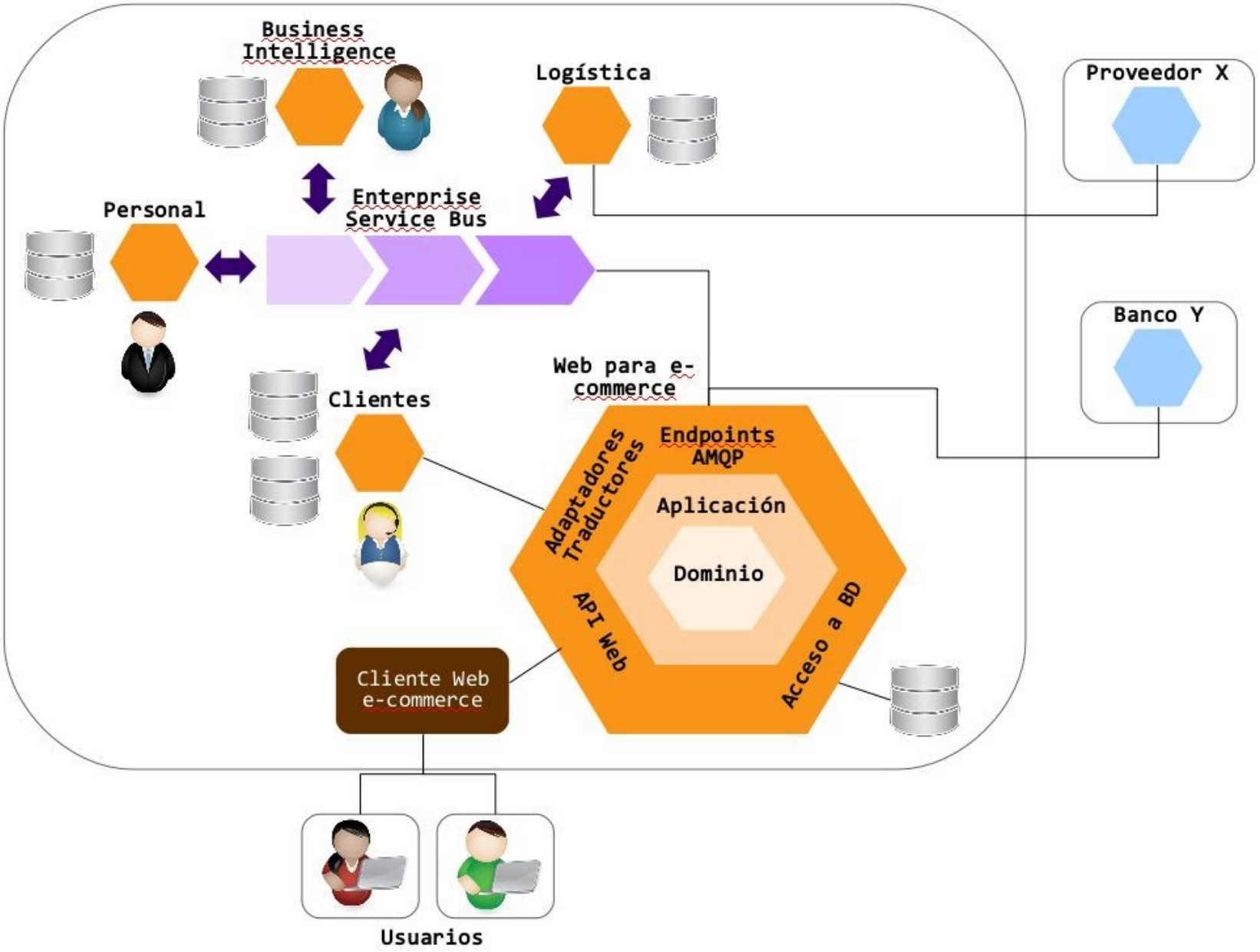
Contextos delimitados

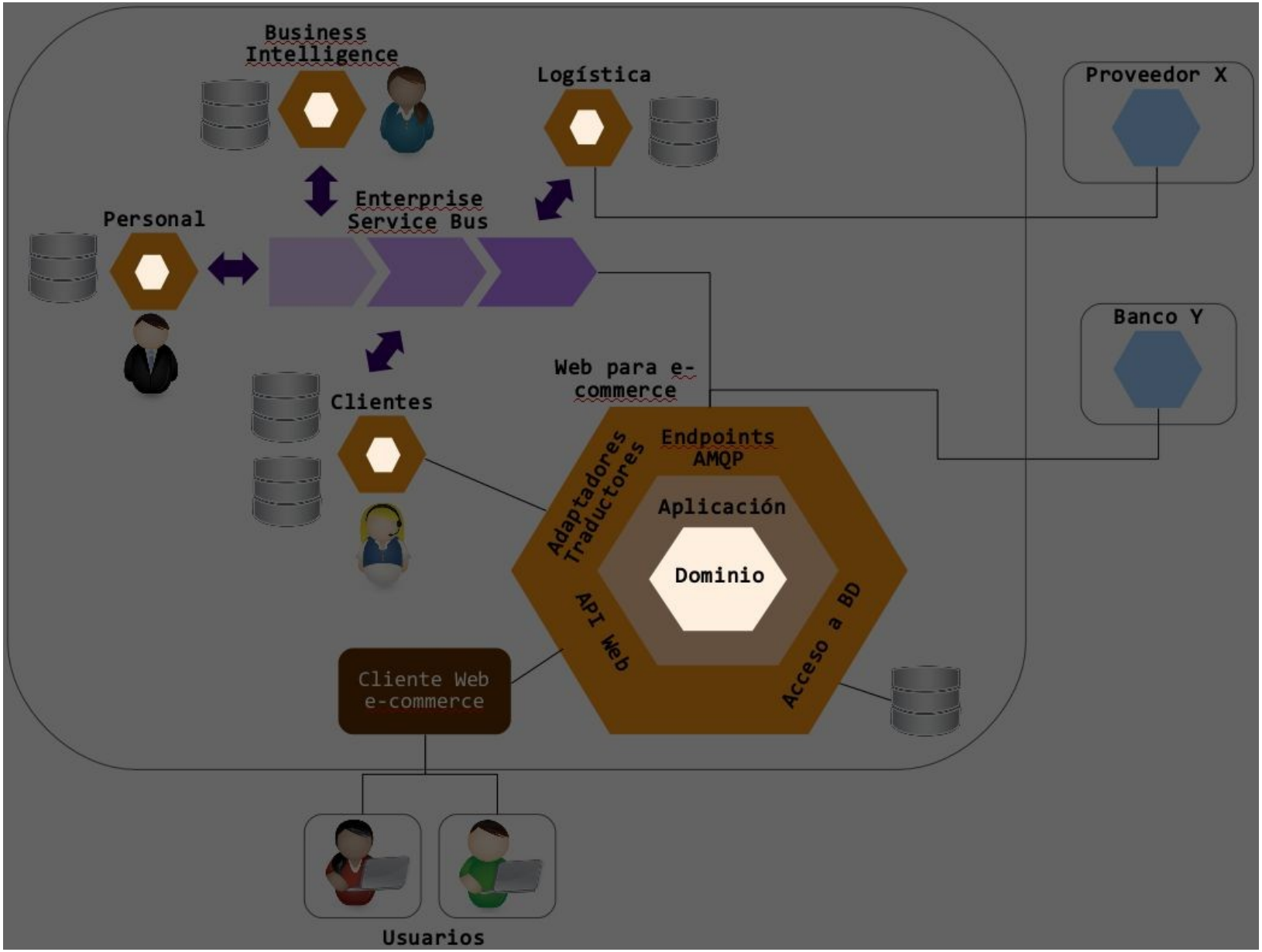
Contexto delimitado

- Un modelo se aplica en cierto contexto determinado, que puede ser:
  - Un sistema completo
  - Un subsistema específico
  - El breve tiempo que dura una discusión de diseño
  - Un microservicio
  - Etc.
- El contexto es el conjunto de condiciones que deben ser ciertas para poder decir que los términos del modelo tienen un significado específico

Contexto delimitado

- Establece explícitamente el contexto en el que cada modelo se aplica
- Establece límites explícitos en términos de organización de equipos de trabajo, uso en partes diferentes de la aplicación, código fuente, esquemas de bases de datos etc.
- Mantén tu modelo estrictamente consistente dentro de estos límites, y no dejes que lo que queda fuera de ellos te confunda o distraiga
  - La interacción entre diferentes contextos requerirá una traducción





Mapa de contextos

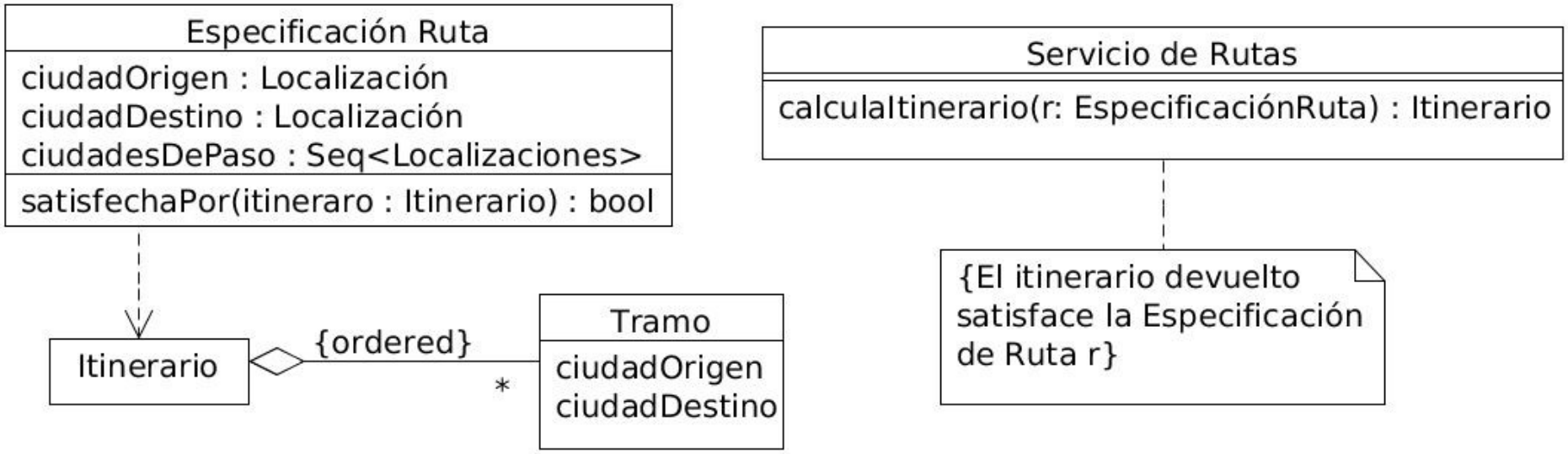
- Los contextos no se pueden ver siempre aisladamente
  - Habrá tareas que requieran combinar funcionalidad y datos de distintos contextos
  - La integración de esta funcionalidad y datos entre contextos debe pasar por puntos explícitos de traducción y adaptación
- Para poder organizar esto es importante definir las relaciones entre los distintos contextos y crear una vista global de todos los contextos en el proyecto
- Esta vista está a medio camino entre la gestión del proyecto y el diseño de software
  - La gente que trabaje junta compartirá de forma natural un contexto
    - Los que no hablen entre ellos no

Mapa de contextos

- Identifica cada modelo en el proyecto y delimita su contexto
- Dale un nombre a cada contexto
  - Este te permite hablar sin ambigüedad del modelo de cualquier parte del sistema, simplemente aclarando cuál es el contexto
  - Ese nombre es parte del lenguaje compartido
- Describe los puntos de contacto entre modelos e indica, a grandes rasgos, traducciones explícitas para cada comunicación
  - Resalta lo que se comparte entre modelos
- La representación del mapa de contextos puede ser gráfica y/o textual
- No te preocupes mucho si detectas problemas o relaciones extrañas entre conceptos, el primer paso es conocer la situación actual

Ejemplo: dos contextos en una aplicación de rutas

- Una aplicación de cálculo de rutas óptimas por carretera que permita visitar una secuencia de ciudades puede tener un modelo de dominio así

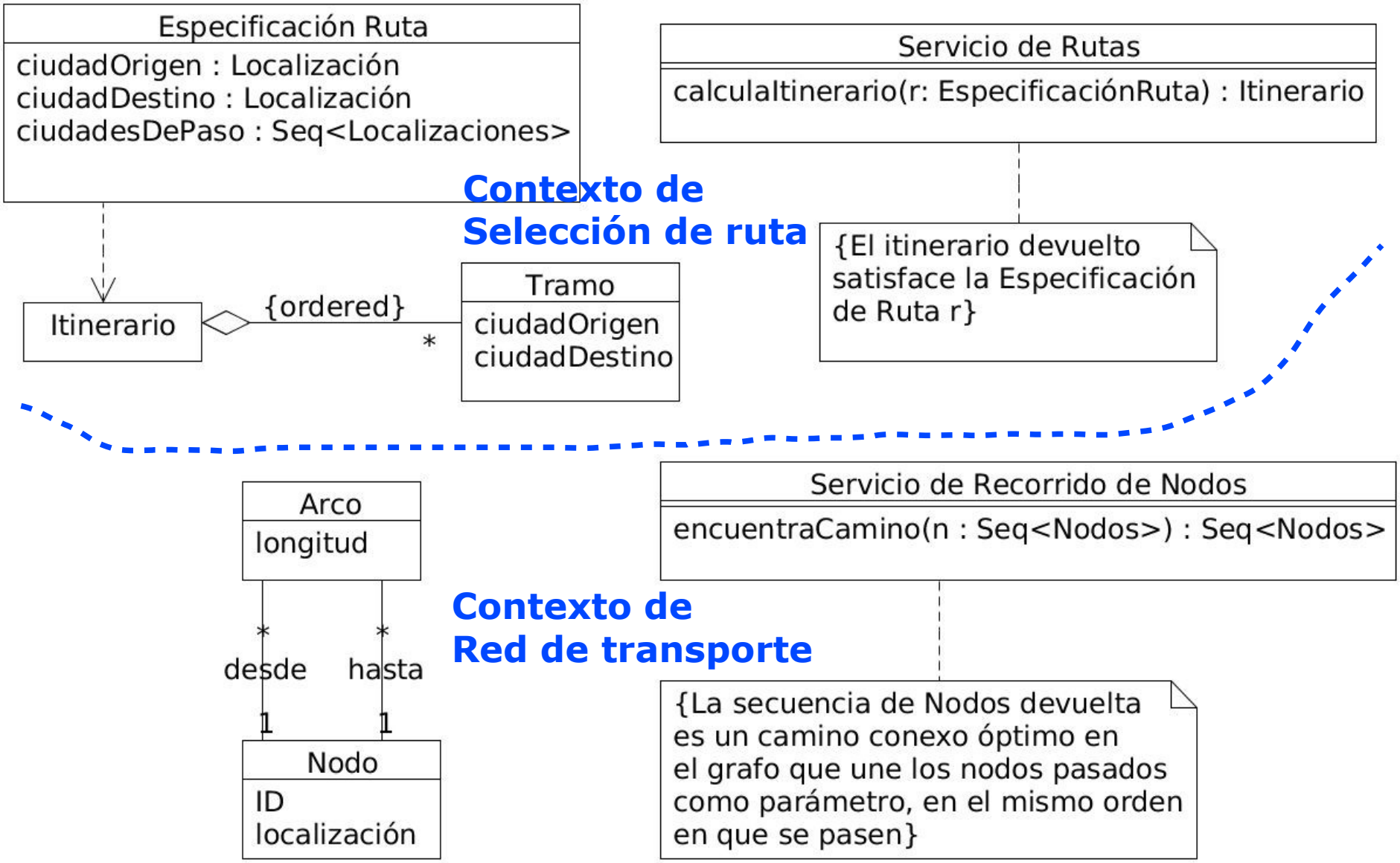


Ejemplo: dos contextos en una aplicación de rutas

- El Servicio de Rutas encapsula un mecanismo posiblemente complejo dentro de una función
  - El resultado se caracteriza mediante aserciones
  - La declaración de la función muestra que cuando se pasa una Especificación de Ruta, se devuelve un Itinerario
  - La aserción indica que el Itinerario devuelto satisfará la Especificación de Ruta pasada

Ejemplo: dos contextos en una aplicación de rutas

- Nuestras rutas consisten en una secuencia de tramos de carretera conectados que nos llevan de una ciudad a otra (pasando por las que le digamos)
  - Necesitamos informar a los usuarios de cómo seguir la ruta
    - "Sigue la N-123 durante 12 km"
  - Y facilitar que puedan hacer ciertas elecciones
    - "Quiero pasar por Villaconejos del Encinar, que dicen que es muy bonito en invierno"
- Podríamos usar el mismo modelo para calcular la ruta óptima
  - Pero queremos buenas prestaciones, y aprovechar algoritmos existentes, y resulta que tenemos un paquete de manejo de grafos muy optimizado que queremos aprovechar
- En resumen, que nos hace falta tener dos modelos de nuestras rutas
  - Uno más expresivo y más cercano al dominio, con objetos conectados
  - Otro más eficiente para poder hacer los cálculos de manera rápida
- Vamos a delimitar dos contextos, uno para cada modelo



Ejemplo: dos contextos en una aplicación de rutas

- Analizando los dos contextos, vemos que no necesitamos mapear todo entre ellos, solo necesitamos dos traducciones específicas
  - Especificación de Ruta → Secuencia de Nodos
  - Secuencia de Nodos → Itinerario
- Para eso hay que entender esos términos en cada modelo y expresarlos en términos del otro modelo

Especificación de Ruta → Secuencia de Nodos

- La especificación de la ruta tiene un origen, un destino y unas ciudades de paso intermedias
  - El origen será el primer nodo de la secuencia, el destino el último y el resto irán entre estos y en orden
- Ambos modelos usan los mismos ids para localizaciones y eso facilita la traducción
- La traducción inversa sería ambigua
  - No todos los nodos del grafo corresponderán con ciudades de paso que el usuario pueda elegir, pueden ser p.ej. cruces entre carreteras

Secuencia de Nodos

origen
ciudadDePaso1
...
destino

Secuencia de Nodos → Itinerario

- El problema consiste en convertir una secuencia de Nodos en una secuencia de Tramos (un Itinerario)
  - Algunos (muchos) de los Nodos no serán relevantes (cruces de carretera...)
- Primero cogeremos los Nodos de la secuencia que son ciudades
  - Tenemos que tener en algún sitio esta información o nuestro sistema no podría funcionar
- Y luego cada uno de estos Nodos será el destino de un tramo y el origen de otro
  - Salvo, claro está, el primero que solo será origen, y el último que solo será destino
  - Respetando el orden en el que estén

Ejemplo: dos contextos en una aplicación de rutas

- Finalmente nos queda implementar esta traducción. Puede ser con un objeto que ofrezca dos servicios
  - Este objeto será el que los desarrolladores que trabajan en cada modelo tienen que mantener conjuntamente
    - Lo demás lo pueden desarrollar por separado sin problemas
  - La responsabilidad de coordinar la interacción entre los dos contextos se la damos al servicio de rutas
- Los servicios son funciones libres de efectos secundarios y, en parte gracias a eso, fáciles de probar
  - Esto es bueno, porque estas interfaces entre contextos hay que probarlas con cuidado
    - Estamos en un punto del sistema donde estamos haciendo encajar modelos dispares, que pueden tener diferencias sutiles que se nos escapen al principio

Ejemplo: dos contextos en una aplicación de rutas

Traductor Ruta-Transporte
convierteEspec(r: EspecificaciónRuta) : Seq<Nodos> convierteCamino(n: Seq<Nodos>) : Itinerario

Y en `Servicio de Rutas:`

```
public static Itinerario calculaItinerario(r: EspecRuta) {
  Seq<Nodos> ciudades = TraductorRutaTransporte.convierteEspec(r);
  Seq<Nodos> nodos =
          ServicioRecorridNodos.encuentraCamino(ciudades);
  Itinerario resultado =
          TraductorRutaTransporte.convierteCamino(nodos);
  return resultado;
}
```

Arquitectura dirigida por eventos

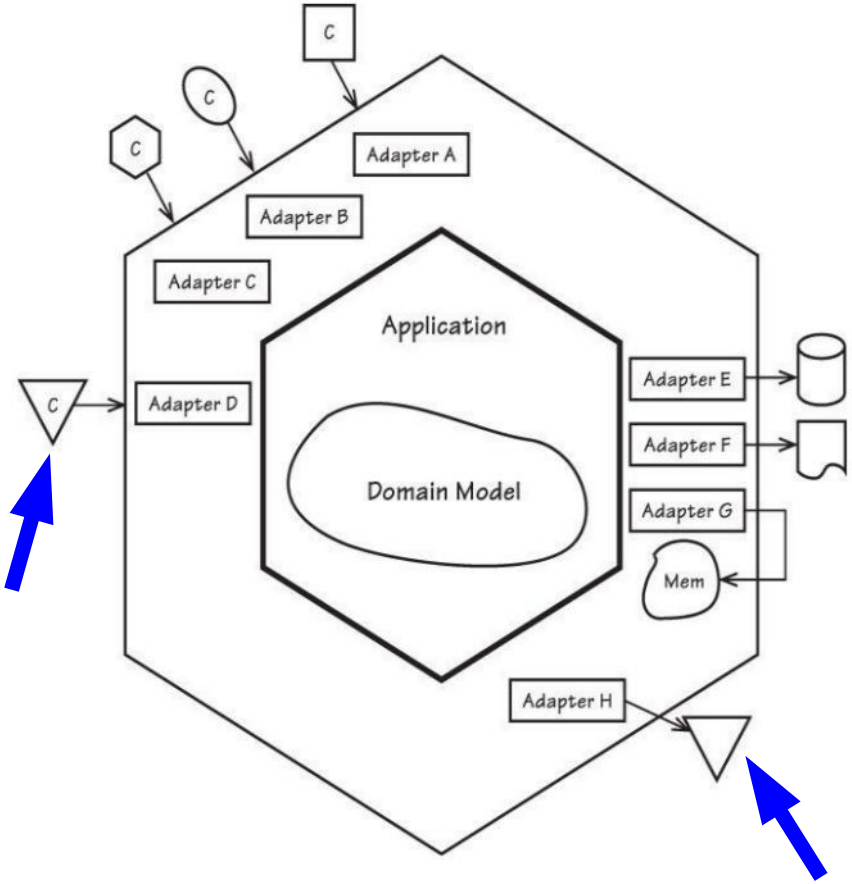
Eventos de dominio

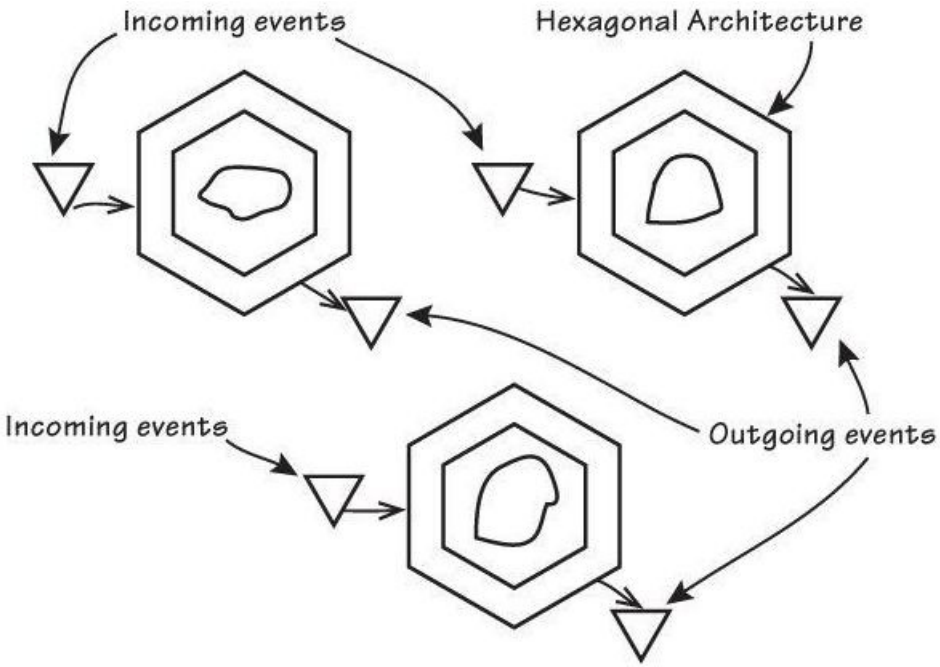
- Un evento de dominio representa la ocurrencia de algo que ha sucedido en el dominio
  - Algo que interesa a los expertos del dominio
  - Normalmente reflejan algo que ocurrió (pasado) y su nombre debería indicar tiempo pasado (`TareaXCompletada`)
- Nos interesa cuándo ocurrió (*timestamp*) y, en general, lo que haría falta si quisiéramos volver a lanzarlo
  - Id del objeto donde tuvo lugar, otros objetos involucrados, algo sobre el estado en el que estaban en ese momento etc.
  - Generalmente los implementaremos como objetos valor inmutables
- Un uso habitual de los eventos de dominio es alcanzar estados consistentes tarde o temprano (o al final) (*eventual consistency*)
  - Por ejemplo para alcanzar consistencia entre elementos distribuidos de manera poco acoplada
- También pueden crearse en respuesta directa a la acción de un usuario

Arquitectura dirigida por eventos

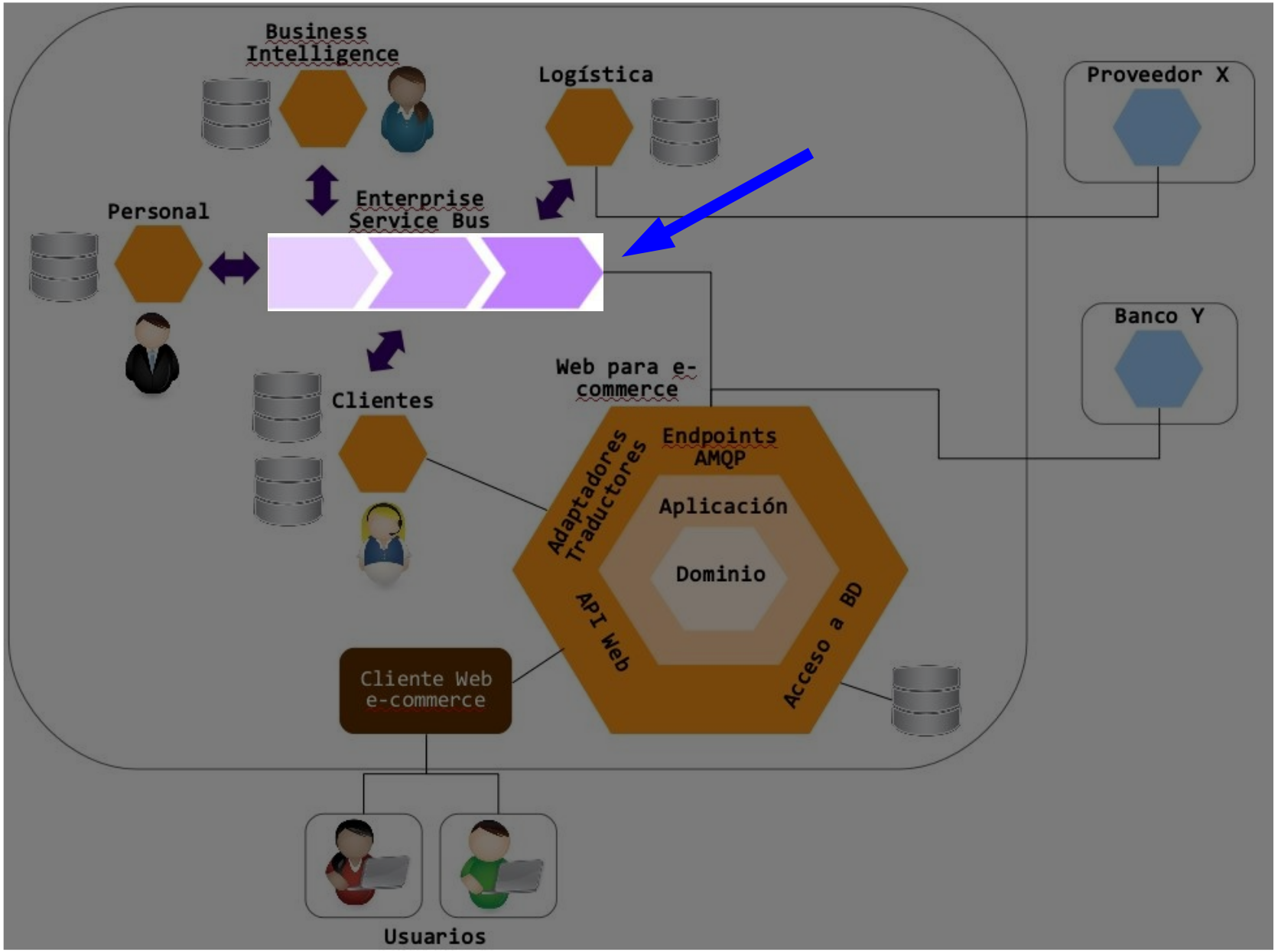
- Una arquitectura dirigida por eventos puede combinar bien con una arquitectura hexagonal
  - Eventos de dominio y también de otros tipos (monitorización, logs etc.)
- Los eventos de dominio publicados por un sistema a través de un puerto de salida llegan a los suscriptores a través de un puerto de entrada
  - Estos eventos tienen un significado específico dentro de cada contexto delimitado al que lleguen
    - Y en algunos contextos quizás ningún significado en absoluto
- Si el tipo es interesante en un contexto, se adaptan sus propiedades a la API de la aplicación (Adaptador) y se ejecuta alguna operación allí
  - Esa operación luego se reflejará en el dominio
  - Recordad que dentro de cada hexágono hay un solo contexto delimitado

Arquitectura dirigida por eventos





© 2013, Vaughn Vernon, Pearson Education, Inc.



Arquitectura dirigida por eventos

- Si un evento tiene que llegar a objetos en un contexto delimitado distinto de aquel donde se originó, normalmente hará falta algún mecanismo de mensajería
  - Por ejemplo algo basado en el protocolo AMQP
  - Dentro de un mismo contexto delimitado se podría recurrir a mecanismos más simples, como el patrón *Sujeto-Observador*
- En todo caso, el estilo de *Publicación-Suscripción* estará en la base de una arquitectura dirigida por eventos

Almacenes de eventos

- Algunos eventos de dominio los vamos a querer guardar
  - Se puede usar para proporcionar luego un *feed* de esos eventos a aplicaciones que prefieren consultarlos a su ritmo (por encuesta)
  - Puede ser un registro histórico de resultados de acciones realizadas sobre el modelo
  - Se puede analizar para ver tendencias, hacer predicciones etc. (*business analytics, business intelligence*)
  - Se puede usar para deshacer cambios en datos
    - Si cada cambio en los datos se publicó como evento, tenemos un registro cronológico de estos cambios
- Si esto nos interesa, normalmente tendremos un repositorio de eventos dentro de cada contexto delimitado para los eventos que se producen allí
  - Ese repositorio se encargará de insertar en la BD a través del adaptador correspondiente

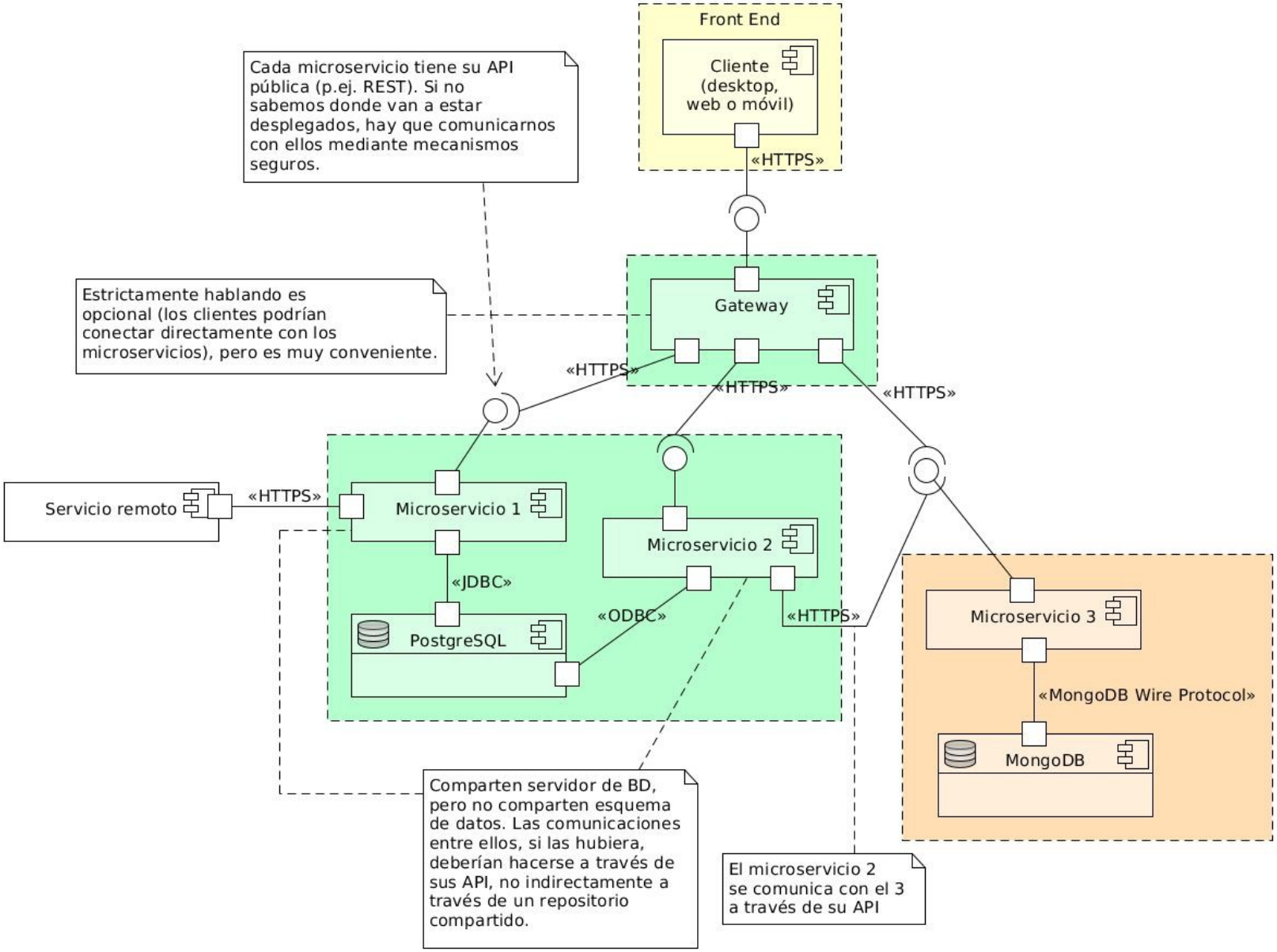
Microservicios

Microservicios

- Aproximación al desarrollo de una aplicación basada en un conjunto de servicios pequeños y poco acoplados, donde cada uno tiene una funcionalidad delimitada dentro del dominio del problema y que se pueden desplegar con independencia
  - Un **conjunto**. Es decir, que no tiene sentido hablar de un microservicio como si pudiera ser una solución a un problema
- Cada microservicio encapsula una parte del dominio del problema
  - Un contexto delimitado (*bounded context*), ni muy pequeño, ni muy grande
    - Muy pequeño: demasiado acoplamiento, demasiadas comunicaciones de red entre microservicios
    - Muy grande: poca cohesión, desarrollo y mantenimiento más complejos
- Cada microservicio tiene su propia base de código, y su desarrollo y mantenimiento lo puede hacer un equipo de desarrollo pequeño
  - Cada microservicio en una misma aplicación puede estar desarrollado con tecnologías, lenguajes, frameworks o bibliotecas distintos
- Cada microservicio proporciona una API definida, típicamente HTTP-REST pero puede ser RPC o mensajería, y esa es la única vía por la que la aplicación y los otros microservicios pueden usar su funcionalidad

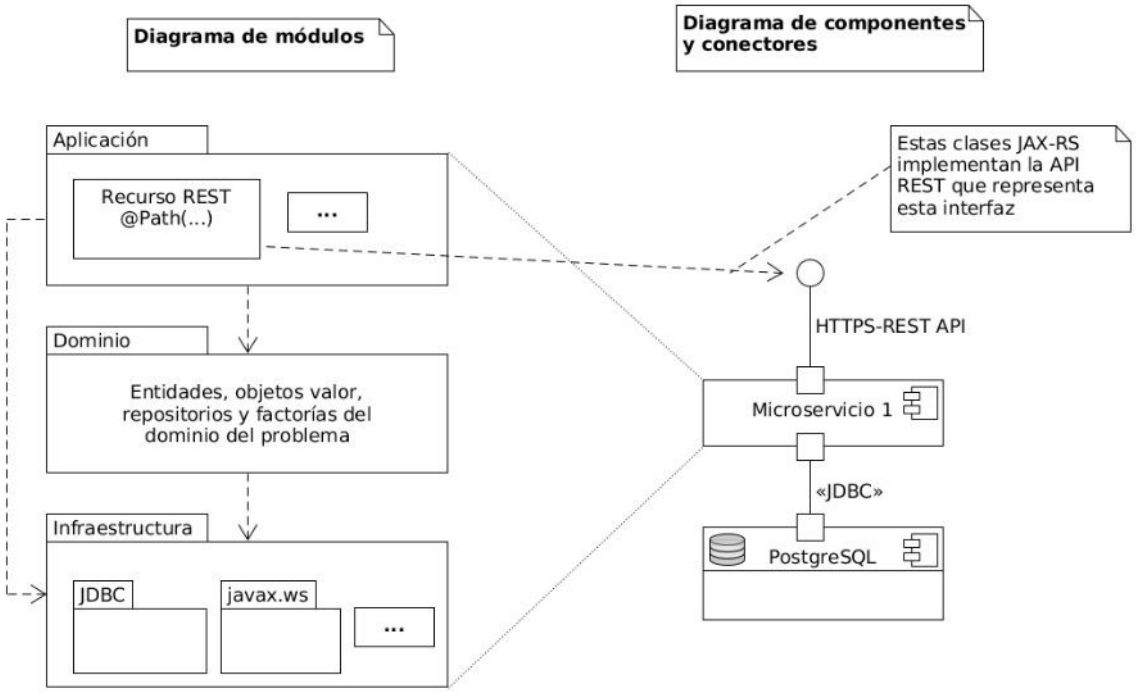
Microservicios

- Cada microservicio es responsable de la persistencia de los datos que necesite
  - No hay un nivel de datos (p.ej. una BD) común a toda la aplicación, y los microservicios no comparten esquemas de datos, tablas etc.
- El despliegue independiente implica que podemos actualizar un microservicio sin reconstruir y desplegar la aplicación completa
- Los clientes podrían llamar directamente a los microservicios que usen, pero es muy conveniente que pasen a través de un gateway
- El equipo que desarrolla cada microservicio suele ser responsable de desarrollar la GUI que le corresponde
  - Puede tener que generar un fragmento de HTML que corresponda a su funcionalidad, y luego un equipo de GUI se encarga de un componente en el lado del servidor, p.ej. en un gateway de agregación, que ensamble esos fragmentos en una plantilla con cierto estilo
  - O bien cada equipo desarrolla un componente de GUI con la región/parte de la pantalla/página que corresponde, y el equipo de GUI del cliente se encarga de crear las plantillas y ensamblarlos



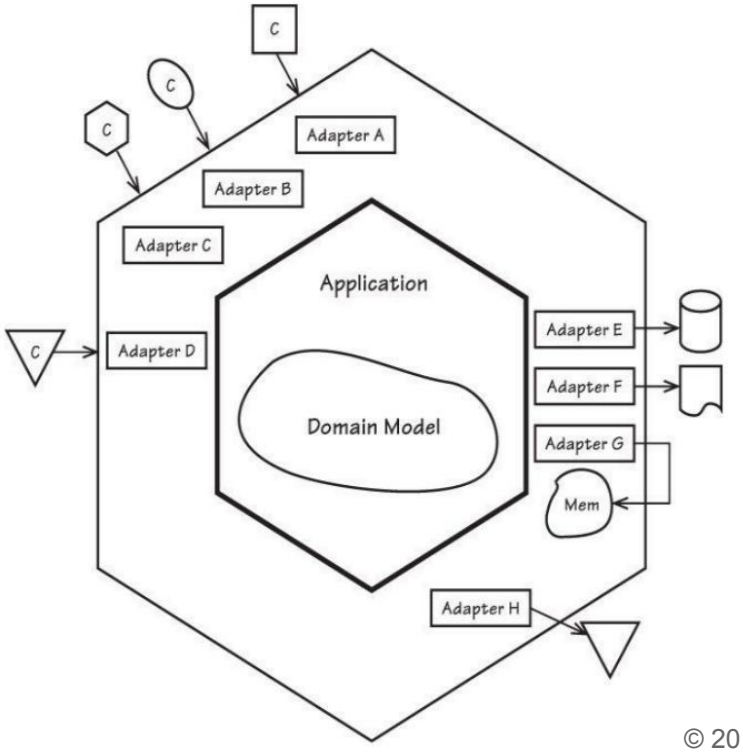
Microservicios – Arquitectura por capas

- El código de cada microservicio puede estar organizado como su equipo de desarrollo prefiera, por ejemplo siguiendo un estilo por capas



Microservicios – Arquitectura hexagonal

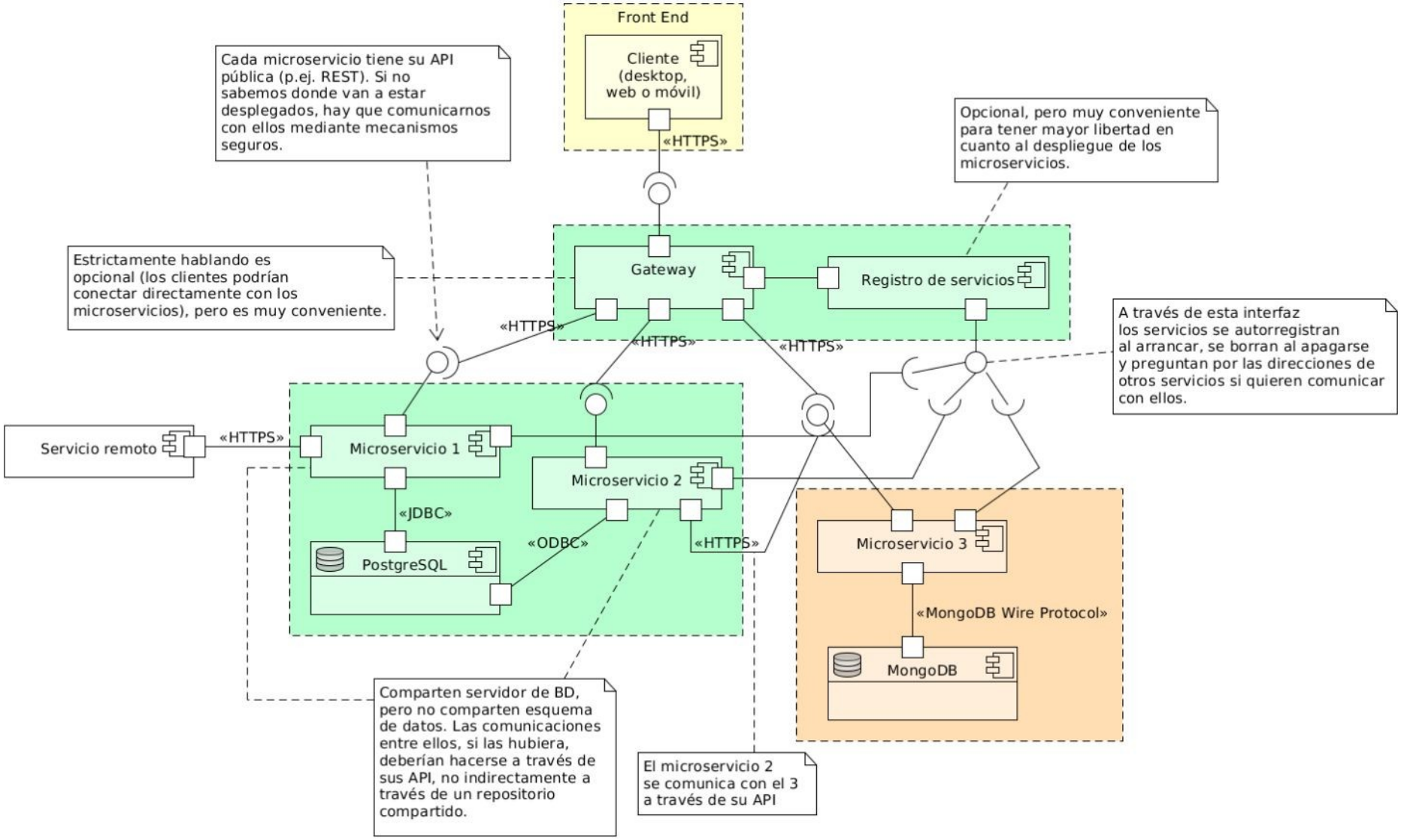
- Pero encaja muy bien usar una arquitectura hexagonal



© 2013, Vaughn Vernon, Pearson Education, Inc.

Microservicios – Registro y descubrimiento

- Normalmente, no siempre, tendremos un registro de servicios
  - Cada tipo de microservicio puede tener varias instancias, desplegadas en sitios distintos, y para cada API que ofrezca necesitamos su localización (host y puerto) para usarla
  - El número de instancias y su localización pueden cambiar dinámicamente
    - Balanceo de carga, despliegues independientes de cada microservicio, IP asignadas dinámicamente etc.
- El registro de servicios permite localizar instancias de los mismos
  - Los servicios se pueden registrar/borrar automáticamente allí cuando son lanzados/parados (*Self Registration*), o lo puede hacer otro componente (*Third Party Registration*)
    - Con el autoregistro tenemos el problema de que un servicio que se ha caído no puede borrarse del registro, además de que cada servicio tiene que tener código para gestionar su propio autoregistro
    - Si hay un tercero que registra y borra los servicios, este es un punto crítico de la infraestructura (y tiene que estar en configuración de alta disponibilidad)



Microservicios – Implementación y despliegue

- Aunque no sea necesario, usar un framework para microservicios puede facilitar la implementación de algunas o todas las piezas (gateway, registro y descubrimiento, configuración externalizada, monitorización, logging…)
  - Spring Boot/Spring Cloud, Vert.x, Go Micro etc.
  - Los proveedores de cloud ofrecen todas las piezas que necesitas y dentro de un mismo proveedor están bien integradas
- Los microservicios se suelen empaquetar cada uno en un contenedor (Docker) para su despliegue
  - También pueden desplegarse en máquinas reales o virtuales, pero eso cada día es menos habitual
  - Empaquetados en contenedores, podemos desplegarlos usando frameworks como Kubernetes o a través de las soluciones cloud como Amazon Elastic Container Service o Google Container Registry

Microservicios – Ventajas

- Puedes actualizar un microservicio sin tener que volver a desplegar toda la aplicación
- Desarrollo independiente
  - Es más fácil que un equipo pequeño focalizado en un microservicio pueda implementar y liberar mejoras con una buena cadencia
  - Cada equipo puede elegir la tecnología que prefiera para su microservicio
- Aislamiento de fallos
  - Si se cae un microservicio, el resto de la aplicación puede seguir funcionando
- Escalado granular
  - Los microservicios se pueden poner en configuraciones de escalado diferentes según su uso, y esto lo controlamos con un grano más fino que lo que podríamos hacer p.ej. con el *middle tier* en una arquitectura *N-tier*

Microservicios – Inconvenientes/Desafíos

- Aunque cada microservicio sea simple, la aplicación en su conjunto será más compleja que si hubiera sido desarrollada como monolito
  - Por ejemplo, dado que cada servicio puede evolucionar por separado hay que tener cuidado con las dependencias entre ellos
- Es complicado probar y cambiar algunas cosas
  - Las dependencias entre servicios son complicadas cuando los servicios evolucionan por separado
  - Tampoco es fácil hacer refactorizaciones cuando estas requieren cambiar más de un servicio
    - Tenemos que sentar a dos o más equipos de desarrollo, que posiblemente usan tecnologías distintas, y ponerlos de acuerdo
- Si tenemos demasiadas tecnologías distintas en la misma aplicación vamos a tener que solucionar algunos problemas varias veces (una vez por cada tecnología que usemos)
  - Es buena idea tener algunos estándares y guías que todo el mundo tenga que seguir, y algunas tecnologías preferidas u obligatorias, especialmente en lo que respecta a temas transversales como logs o seguridad

Microservicios – Inconvenientes/Desafíos

- Vamos a tener más tráfico de red que con una aplicación menos distribuida
- La integridad de los datos es más compleja de mantener
  - Las transacciones distribuidas son más complicadas y poco eficientes
  - Normalmente tendremos que aceptar menos garantías en los cambios en los datos que involucren a varios microservicios
- Generalmente no se recomienda usar microservicios en una organización que no haya internalizado prácticas de *devops*
  - En los microservicios el desarrollo y el despliegue van muy entrelazados

Microservicios – ¿Cuándo usarlos?

- Teniendo en cuenta sus ventajas y sus inconvenientes, podemos considerarlos para
  - Aplicaciones grandes en las que hay que sacar actualizaciones con una frecuencia alta
  - Aplicaciones complejas que necesitan ser muy escalables (en horizontal)
  - Aplicaciones con dominios de problema ricos, o con muchos subdominios de problema distintos
  - Organizaciones que ya se organizan en equipos de desarrollo pequeños, y que manejan adecuadamente buenas prácticas de *devops* incluyendo integración/despliegue continuos

Microservicios y DDD

- Varias herramientas y conceptos del DDD son muy útiles para el correcto diseño de un sistema basado en microservicios
  - Cada microservicio se debería diseñar para ofrecer la funcionalidad relacionada con el modelo de un único contexto delimitado
  - La arquitectura hexagonal es un buen punto de partida para la arquitectura de cada microservicio
  - Muchas de las comunicaciones entre microservicios en general las preferiremos basadas en mensajes
    - En un caso claro de arquitectura dirigida por eventos
      - Eventos encapsulados en mensajes que llegarán a, y saldrán a través de, alguno de los puertos de cada microservicio si este sigue la arquitectura hexagonal

Microservicios y DDD

- Cada microservicio debería corresponder con un contexto delimitado, no hay que tener un microservicio para cada entidad o agregado
  - Eso nos complicaría el mantenimiento de los modelos de dominio al dificultar hacer cambios en los mismos cuando involucran a distintas entidades
  - Además acabaríamos pagando esta hiperfragmentación en prestaciones (más tráfico de red) y en complejidad de despliegue y operaciones
    - Porque tendremos muchas historias de usuario que involucren a varias entidades y agregados
- Los microservicios son una buena forma de separar los aspectos del sistema que requieren un modelo de dominio rico de los que no
  - P.ej. podemos tener microservicios que hagan tareas simples (p.ej. solo lectura de datos, generación de reports etc.) implementados con una sencilla capa de acceso datos en lugar de con una capa de dominio tipo DDD
    - Incluso eligiendo para ellos un framework de persistencia más sencillo y eficiente

Bibliografía

- Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*
  - Capítulo 14
- Vaughn Vernon. *Implementing Domain-Driven Design*
  - Capítulos 4 y 8
- Sam Newan. *Building Microservices. Designing Fine-Grained Systems*