# 5 Domain Strategic

Del dominio táctico al diseño estratégico del negocio.

## 1. CONNECT

- ¿Todo el dominio es igual de importante?
- ¿Qué parte del código refleja realmente la ventaja competitiva?
- ¿Cómo se organizan equipos y módulos alrededor del negocio?

> Objetivo: entender por qué necesitamos **subdominios** y **bounded contexts** para que el modelo de dominio escale sin caos.

# 2. CONCEPT

## Conceptos clave de DDD estratégico

- **Dominio**: el problema de negocio completo que queremos resolver.
- **Subdominios**:
  - Partes del dominio con una responsabilidad de negocio clara.
    - **Core**: ventaja competitiva, máxima inversión.
    - **Supporting**: necesarios pero no diferenciales.
    - **Generic**: comodities, normalmente comprables o externalizables.

- **Bounded Context**:
  - Límite explícito donde un modelo es **coherente y consistente**.
  - Dentro hay un lenguaje **ubicuo** claro y reglas homogéneas.
  - Fuera, las mismas palabras pueden significar otra cosa.

## Relaciones entre contextos

- **Context Map**:
  - Diagrama que muestra **qué contextos existen** y **cómo se relacionan**.
- Tipos de relaciones frecuentes:
  - **Customer / Supplier**: un contexto depende del modelo de otro.
  - **Conformist**: un contexto se adapta sin cuestionar al proveedor.
  - **Anticorruption Layer (ACL)**: un contexto se protege del modelo del otro mediante un traductor.

> No existe un **modelo único perfecto**

# 3. CONCRETE PRACTICE

Partimos de una con arquitectura hexagonal y módulos técnicos.

Vamos a **refactorizar hacia módulos alineados con el negocio**:

- Subdominio de **Fleet** (gestión de cohetes y vuelos).
- Subdominio de **Sales** (reservas y cancelaciones).
- Kernel **Shared** con utilidades y conceptos transversales.

> Objetivo: separar físicamente el código en subdominios y definir contratos explícitos entre ellos.

1. **Crear estructura de paquetes**
   - Crear paquetes raíz:
     - `com.astrobookings.fleet`
     - `com.astrobookings.sales`
     - `com.astrobookings.shared`
2. **Mover kernel compartido**
   - Mover excepciones (`BusinessException`, `BusinessErrorCode`) y utilidades comunes a `com.astrobookings.shared`.

3. **Migrar subdominios**
   - Mover a `fleet`:
     - Entidades `Rocket`, `Flight`, `FlightStatus`, puertos `RocketRepository`, `FlightRepository` y servicios `RocketsService`, `FlightsService` a `fleet.domain`
     - Adaptadores a `fleet.infrastructure`
   - Mover a `sales`:
     - Entidad `Booking` a `sales.domain` Puertos `BookingRepository` y servicio `BookingsService` a `sales.domain`
     - Adaptadores a `sales.infrastructure`

4. **Definir contrato de comunicación (Sales → Fleet)**
   - En `sales.domain.ports.out`, crear una interfaz `FlightInfoProvider` (o `FlightGateway`) con:
     - `getFlightStatus(String flightId)`
   - Objetivo: que `BookingsService` **no dependa** directamente de `FlightRepository` de `fleet`.
5. **Implementar adaptador de comunicación**
   - En `sales.infrastructure`, crear `FlightAdapter` que implemente `FlightInfoProvider`.
   - Este adaptador inyectará `fleet.*` (repositorios o uses cases) para obtener datos.

6. **Refactorizar `BookingsService` y `CancellationService`**
   - Usar la dependencia de `FlightRepository` para
     - obtener datos de los vuelos.
     - actualizar los estados de los vuelos.
7. **Recablear la aplicación (composition root)**
   - Actualizar `Config.java` y `AstroBookingsApp.java`.
   - Instanciar primero el módulo `Fleet`.
   - Instanciar el módulo `Sales` inyectándole el adaptador que conecta con `Fleet`.

# 4. CONCLUSIONS

- Separar el código en **subdominios** y **bounded contexts** reduce el acoplamiento organizativo y técnico.
- Los contratos explícitos (interfaces entre contextos) permiten que cada módulo evolucione a su ritmo.
- La pregunta deja de ser solo *"¿qué capa toca esto?"* y pasa a ser:
  - *"¿en qué subdominio / contexto vive esta decisión de negocio?"*

> ¿Qué subdominios y bounded contexts identificarías mañana mismo en tu sistema?