6 Domain Tactic

Del diseño estratégico a la implementación táctica rica.

1. CONNECT

- ¿Tus clases de dominio son solo "bolsas de datos" (getters/setters)?
- ¿La lógica de negocio está dispersa en "Servicios" gigantes?
- ¿Validamos lo mismo en 3 sitios diferentes?

> Objetivo: pasar de un **Modelo Anémico** a un **Modelo Rico** que proteja sus invariantes.

2. CONCEPT

Building Blocks del DDD Táctico

- **Value Objects (VO)**:
  - Se definen por sus atributos, no tienen identidad.
  - Son **inmutables**.
  - Ejemplo: `Money`, `Email`, `GPSLocation`.
- **Entities**:
  - Tienen **identidad** única que perdura en el tiempo.
  - Tienen ciclo de vida (cambian de estado).
  - Ejemplo: `Flight`, `Booking`.

- **Aggregates**:
  - Conjunto de entidades y VOs tratados como una **unidad de consistencia**.
  - Tienen una **Root Entity** (la única accesible desde fuera).
  - Regla de oro: Una transacción = Un Agregado.
- **Domain Services**:
  - Lógica de negocio que no pertenece naturalmente a una entidad/VO.
  - Orquestan operaciones entre varios agregados.
  - No confundir con *Application Services* (que solo orquestan casos de uso).

Anemic vs Rich Model

- **Anemic Model**:
  - Entidades con solo datos + Setters públicos.
  - Lógica en Servicios externos.
  - "Anti-patrón" en DDD (aunque común en otros estilos).
- **Rich Model**:
  - Entidades con **comportamiento**.
  - Métodos con nombres de negocio (`flight.cancel()`, no `flight.setStatus(...)`).
  - La entidad se autovalida y protege su estado.

3. CONCRETE PRACTICE

Vamos a aplicar DDD táctico en pequeño, sobre `Fleet` y `Sales`.

1. Enriquecer `Flight` (Fleet)

- **Qué detectar**:
  - Un vuelo hoy solo "cambia de estado" desde servicios.
  - La lógica de confirmar, cancelar o llenar el vuelo está fuera de la entidad.
- **Qué hacer**:
  - Añadir métodos de negocio en `Flight` (por ejemplo: `confirm(...)`, `cancelDueToLowDemand(...)`, `markSoldOut()`).
  - Encapsular los cambios de `FlightStatus` para que no se hagan con setters desde fuera.

2. Enriquecer `Booking` (Sales)

- **Qué detectar**:
  - `Booking` se construye con setters y puede quedar en estados inválidos.
- **Qué hacer**:
  - Crear un método de factoría `Booking.create(...)` que valide los datos de entrada.
  - Asegurar que una reserva creada siempre tenga un estado coherente (precio válido, vuelo asociado, etc.).

3. Introducir el VO `Capacity`

- **Qué detectar**:
  - La capacidad máxima del cohete se valida en varios sitios con primitivos.
- **Qué hacer**:
  - Crear el Value Object `Capacity` para encapsular la capacidad del cohete y sus reglas (rango permitido).
  - Usar `Capacity` en `Rocket` (y donde aplique en `Flight`) en lugar de un `int` suelto.

4. Refinar Servicios de Dominio

- **Qué detectar**:
  - `BookingsService`, `FlightsService` y `CancellationService` contienen mucha lógica de dominio detallada.
- **Qué hacer**:
  - Dejar en los servicios solo la orquestación (llamadas a repositorios, gateways, notificaciones).
  - Delegar en `Flight`, `Booking` y `Capacity` las reglas de negocio que afectan a su propio estado.

> En este taller implementaremos de forma concreta un único Value Object táctico: `Capacity`, para encapsular la capacidad de los cohetes y reutilizar sus invariantes. Otros posibles VOs (fechas de vuelo, precios, descuentos) se mencionan solo como inspiración.

> **Regla**: No modificar `Flight` y `Booking` en la misma transacción de base de

4. CONCLUSIONS

- El **Modelo Rico** encapsula la lógica donde pertenecen los datos.
- Los **Value Objects** eliminan la obsesión por los primitivos y reducen bugs.
- Los **Agregados** definen límites claros para la consistencia de datos.
- Los **Servicios de Dominio** son para lo que "no cabe" en las entidades, no para "toda la lógica".

> ¿Qué entidad de tu sistema actual pide a gritos tener comportamiento propio?