Patrones de diseño aplicables en la arquitectura

Con la estructura actual del proyecto, existen **patrones de diseño** que ya se aplican implícitamente y otros que pueden incorporarse de forma natural. A continuación se listan por categoría, mapeando a rutas del árbol, el problema que resuelven y fragmentos de código ilustrativos.

---

1. Patrones arquitecturales

1.1. Capas (Layered Architecture)

Qué resuelve: separa Interfaz → Aplicación → Dominio → Infraestructura.
Evidencia:

- Interfaz: `domains/user/api/routes,controllers`
- Aplicación: `domains/user/application/commands,queries,services`
- Dominio: `domains/user/domain/entities,ports`
- Infraestructura: `infrastructure/*/models,mappers,repositories`

1.2. Puertos y Adaptadores (Hexagonal)

Qué resuelve: el dominio depende de **interfaces** (puertos); las implementaciones viven en **infraestructura**.
Evidencia:

- Puertos: `domains/user/domain/ports/ex.ts` (placeholder)
- Adaptadores: `infrastructure/user/repositories/ex.ts`

1.3. CQRS (Command–Query Responsibility Segregation)

Qué resuelve: separa lectura (*queries*) de escritura (*commands*).
Evidencia: `domains/user/application/queries/ex.ts`, `domains/user/application/commands/ex.ts`.

2. Patrones estructurales

2.1. Repository

Qué resuelve: oculta los detalles de acceso a datos tras una interfaz de dominio.
Evidencia:

- Interfaz (puerto): `domains/user/domain/ports/ex.ts`
- Implementación (infra): `infrastructure/user/repositories/ex.ts`

Ejemplo (puerto):

```
// domains/user/domain/ports/UserRepository.ts
export interface UserRepository {
save(u: User): Promise<void>;
findById(id: string): Promise<User | null>;
}
```

2.2. Data Mapper

Qué resuelve: mapea *entidad de dominio ↔ modelo ORM/ReadModel.*
Evidencia: infrastructure/user/mappers/ex.ts
Ejemplo:

```
// infrastructure/user/mappers/UserMapper.ts
export const toReadModel = (row: UserModel) => ({ id: row.id, name: row.
    name });
```

2.3. Adapter (HTTP → Caso de uso)

Qué resuelve: adapta el mundo HTTP (Express) al contrato de la aplicación.
Evidencia: domains/user/api/controllers/ex.ts, domains/user/api/routes/ex.ts

2.4. DTO (Data Transfer Object)

Qué resuelve: contratos de entrada/salida estables y serializables.
Evidencia: domains/user/schemas/ex.ts, shared/domain/base$_{r}esponse.ts$

3. Patrones creacionales

3.1. Factory Method / Static Factory (en Entidades)

Qué resuelve: creación controlada con **invariantes** del dominio.
Evidencia: domains/*/domain/entities/*.ts (placeholders; ideal para aplicar).
Ejemplo:

```
// domains/user/domain/entities/User.ts
export class User {
private constructor(private p: Props) {}
static create(p: Props) {
if (!p.email.includes("@")) throw new Error("email inv lido");
return new User({ ...p, createdAt: new Date() });
}
}
```

3.2. Abstract Factory (opcional)

Qué resuelve: elegir repos/infra por entorno (p.ej., Postgres vs. MySQL).
Evidencia: no explícita; puede orquestarse desde core/dependencies/dependencies.ts.

4. Patrones de comportamiento

4.1. Chain of Responsibility (Middlewares)

Qué resuelve: pipeline de responsabilidades desacopladas.
Evidencia: core/middlewares/requestValidator.ts,errorHandler.ts,responseInterceptor.ts,respons
Ejemplo conceptual:

```
app.get("/v1/users", requestValidator(schema), controller,
    responseInterceptor);
```

4.2. Strategy (Validación / Logging)

Qué resuelve: intercambiar algoritmos en tiempo de ejecución (validadores, formateadores de log).

Evidencia:

- Validación: core/middlewares/requestValidator.ts + domains/user/schemas/ex.ts
- Logging: core/logging/helpers/formatHttpLoggerResponse.ts,timeStampFormat.ts,sensitiveInfoEx

4.3. Template Method (Base Service)

Qué resuelve: define el esqueleto de un flujo y delega pasos a subclases.
Evidencia: shared/domain/base$_service.ts(contenidonovisible; supuestoaplicable)$.
Ejemplo de intención:

```
// shared/domain/base_service.ts
export abstract class BaseService<I,O>{
async execute(i: I){ this.validate(i); const o = await this.run(i); return
    this.wrap(o); }
protected validate(i: I){ /* opcional */ }
protected abstract run(i: I): Promise<O>;
protected wrap(o: O){ return { ok: true, data: o }; }
}
```

4.4. Interceptor

Qué resuelve: transformación/observación de la respuesta sin tocar el controller.
Evidencia: core/middlewares/responseInterceptor.ts

4.5. Observer / Domain Events (recomendado)

Qué resuelve: reacciones desacopladas a hechos del dominio.
Evidencia: no se observa un Event Bus en el árbol (supuesto). Cómo incorporarlo: core/events/EventBus.ts + application/subscribers/* por dominio.
Ejemplo mínimo:

```
// core/events/EventBus.ts
type Handler = (e: { type: string; payload: any }) => Promise<void> | void;
export class EventBus {
private map = new Map<string, Handler[]>();
on(type: string, h: Handler){ this.map.set(type, [...(this.map.get(type)
    ||[]), h]); }
async emit(e:{type:string;payload:any}){ for(const h of this.map.get(e.type
    )||[]) await h(e); }
}

// domains/exam-generation/application/commands/GenerateExamHandler.ts (
    idea)
await repo.save(exam);
await eventBus.emit({ type: "ExamGenerated", payload: { examId: exam.id }
    });

// domains/exam-application/application/subscribers/AssignOnExamGenerated.
    ts
eventBus.on("ExamGenerated", async (e) => { await assignExamHandler.execute
    ({ examId: e.payload.examId }); });
```

4.6. Specification (opcional)

Qué resuelve: combinaciones reutilizables de criterios (AND/OR) para consultas.
Evidencia: no explícita; buen encaje en domains/*/application/queries con domains/*/schemas.

4.7. Unit of Work (transacciones)

Qué resuelve: atomizar múltiples operaciones de repositorio bajo una transacción.
Evidencia: database/database.ts (bootstrap). No se ve UoW explícito; puede exponerse un withTransaction(fn) desde database/* y usarse en handlers.

5. Cómo aterrizar 5 patrones ya

1. **Repository + Mapper (reforzar):** garantizar puertos en domains/*/domain/ports y adaptadores en infrastructure/*/repositories + infrastructure/*/mappers.
2. **Factory Method en entidades:** añadir static create(...) en domains/*/domain/entities/*.ts con invariantes.
3. **Chain of Responsibility (ya):** usar requestValidator + responseInterceptor + errorHandler en todas las rutas.
4. **Strategy en logging (ya):** estrategias en core/logging/helpers/*; configurar el logger para aceptar formatter/masker.
5. **Template Method en servicios:** estandarizar handlers con validate → run → wrap.

Resumen

- **Ya presentes:** Layered, Ports & Adapters, Repository, Data Mapper, DTO, Adapter (HTTP→App), Chain of Responsibility, Strategy, Interceptor.
- **Fáciles de incorporar:** Factory Method (entidades), Template Method (base$_service$), $UnitofWork(da$

- **Beneficio:** refuerzan separación de responsabilidades, testabilidad, sustituibilidad de infraestructura y evolución por dominios sin romper el núcleo.