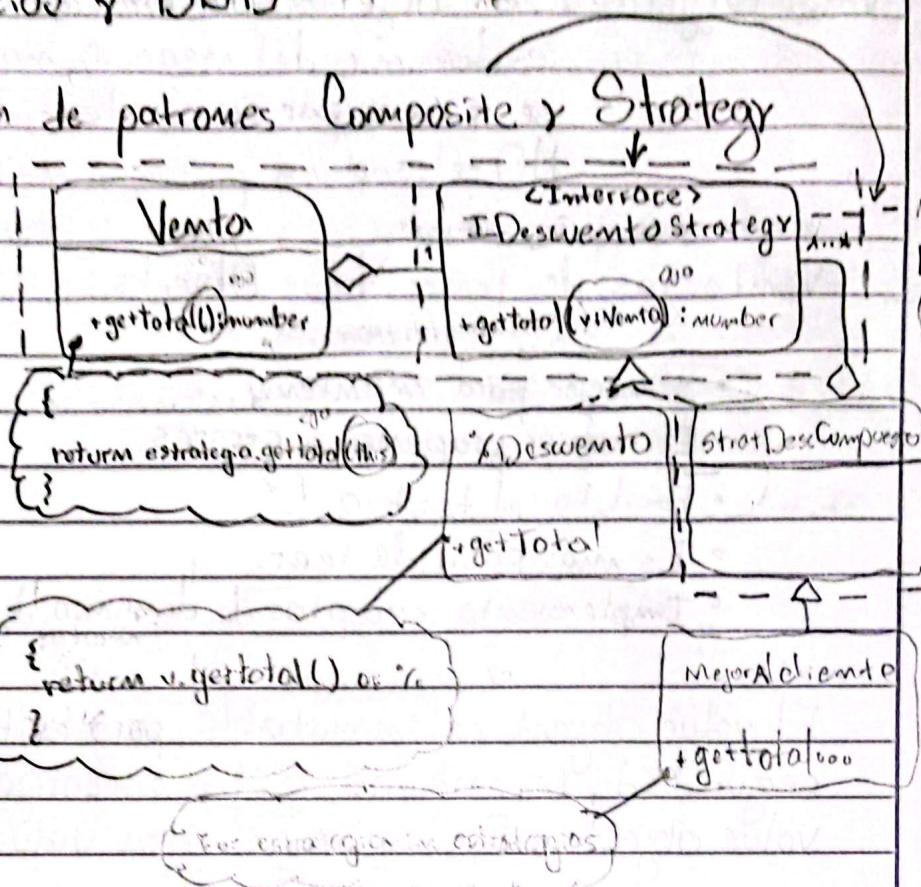


Clase #12 - Ejercicios y DDD

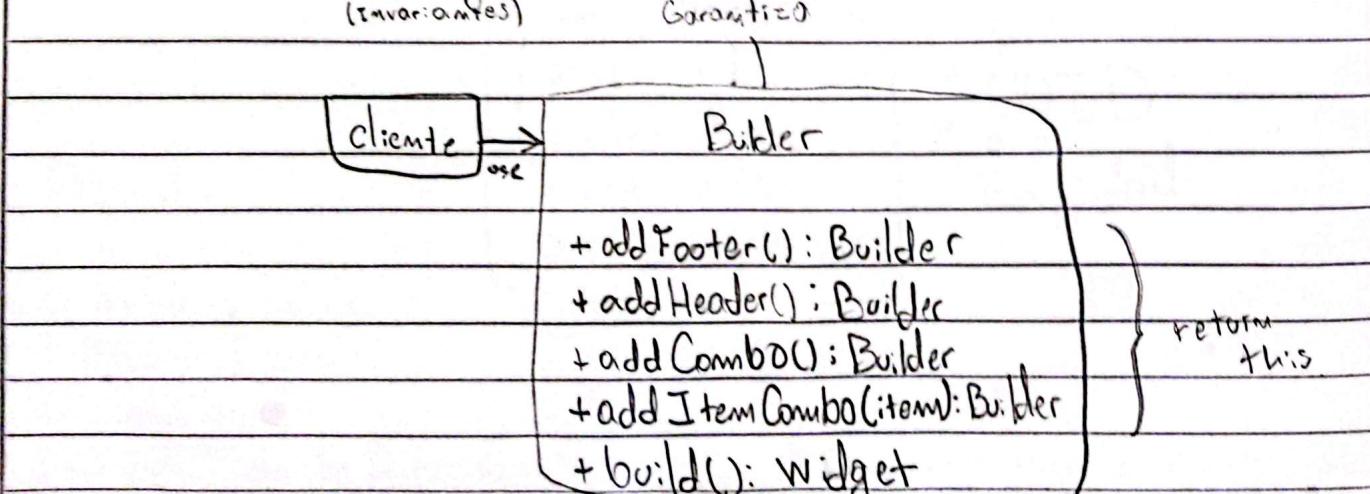
Ejercicio: Composición de patrones Composite y Strategy

- ① Pase de parámetros entre mensajes que intercambian objetos.

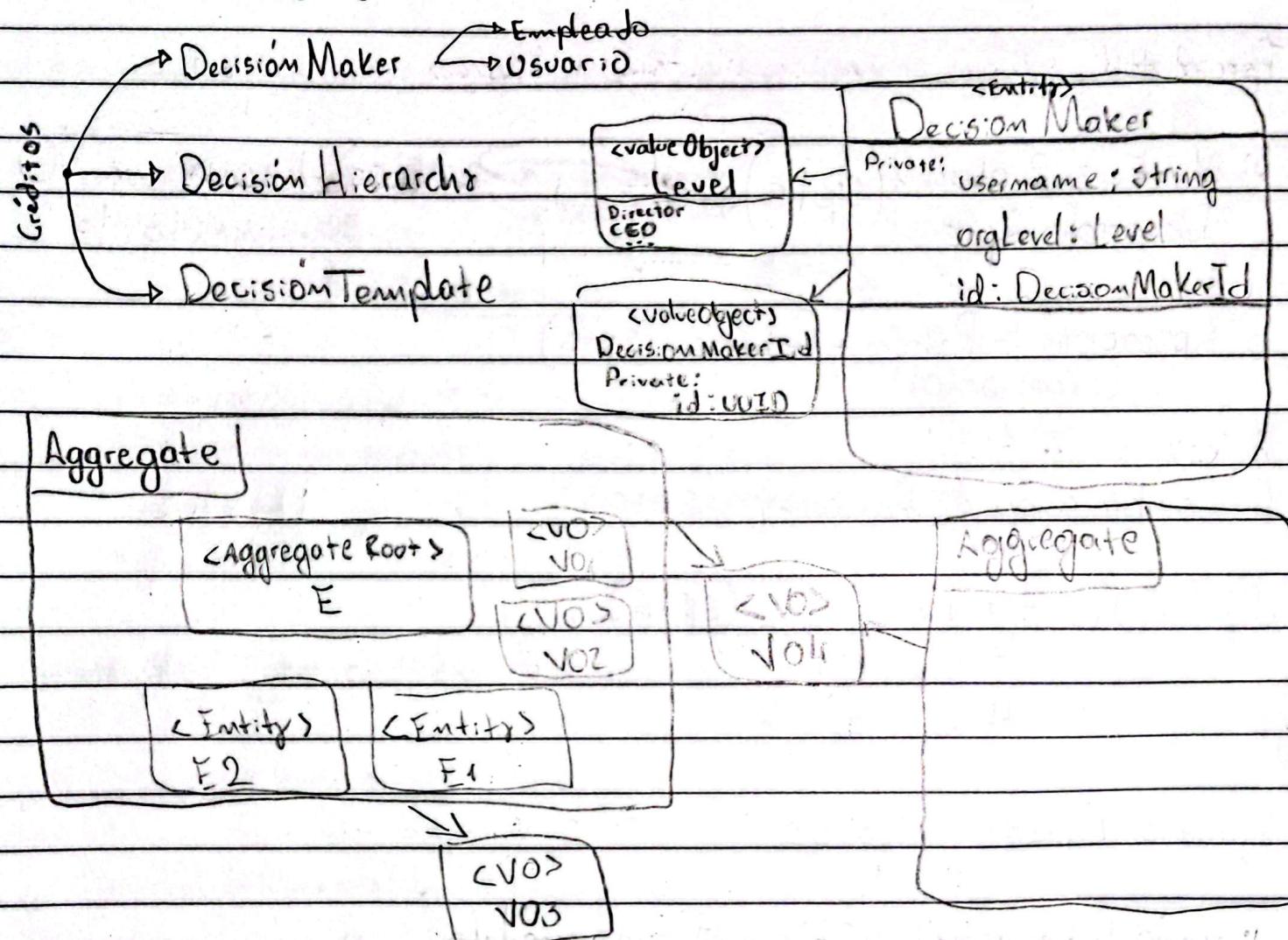


Ejercicio: Builder

- ① Encapsular {
 ↗ new
 ↗ representaciones futuras
 ↗ lógica de creación y sus reglas
 ↗ invariantes } Garantizo



DDD: Aggregate (Ejemplo del préstamo crediticio)



Clase #13 - Antes del Parcial

Parcial #1

a) V/F 3 pts } (5pts) → Valor de que un VO
Desarrollo 2 pts sea immutable.

b) Aplicación de Patrón + Genérico (5pts)
(Creacionario)

c) Combinación de Patrones (5pts)

d) DDD → VO (5pts)

 → Entidades

 → Relaciones

Ejercicio Tickets - Notas

- Todo lo referente a tiempo se models.
- Los archivos no son VO, son Entidades.
- Un mensaje en un correo puede ser VO, pero aquí es Entidad
- Se puede tener un historial si algo puedes cambiar y quieres mantenerlo en algún sitio.

Clase #17 - Agregados

Aggregates

¿Qué los caracteriza?

Falso

- Reglas de Negocio → Restricciones
- Invariantes
- SOLID → SRP
- Frontera de Consistencia

Verdad

Por agrupar muchas entidades y VO

El agrega lo nunca
puede tener un estado
inválido.

Método de Pago
es un agregado.

Va a velar por la
consistencia e
invariantes.

Clase #19 - Ejercicio en clases

(Servicio de Dominio)

• Asignar tarea a Dev.

146 horas/mes max

• Tarea (Entity)

- Tiempo Estimado
- Estado
 - To Do
 - In Progress
 - Complete

• Developer (Entity)

- Rango
 - Junior
 - Semi-Senior
 - Senior
- Tiempo
 - Ocupado
 - Disponible

• Aspectos

- ↳ Log
- ↳ Exito
- ↳ Errores
- ↳ Performance

class AsignarTarea {

execute(Devs:Developer[], task:Tarea): Either<E,D>

for (dev:in Devs){

if (dev.rango.equals(-) && sumahoras ≤ 100){

return Either.Right<E,D>(dev);

E = error

D = Developer

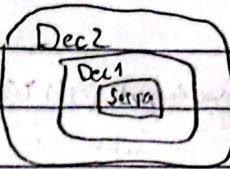
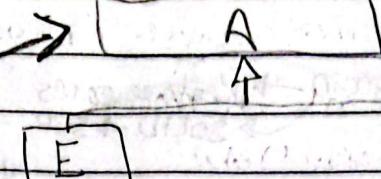
Clase #20 - Asignar responsabilidades

Aspectos (AOP)

↓
Aspectos

↓
Modelado con
Decorador

(Interface)



Hexagonal + AOP

Tipos Parametrizados

< TService, RService >

No recibe información
en sus tipos.

Aspecto

La lógica de los aspectos debe estar lo menos
anclada a sus tipos parametrizados.

Logger de Errores

```
class LoggingServiceDecorator<TS, RS> implements IService<TS, RS> {
    private service: IService<TS, RS>; logger: ILogger;
    execute(s: TS): Either<Error, RS> {
        const r = this._service.execute(s);
        if (r.isLeft()) {
            this.logger.error(`_____ + ${r.getLeft()}`);
        } else {
            this.logger.debug(`_____ + ${s} + ${r.getRight()}`);
        }
        return r;
    }
}
```

↳ Trucado por el
lenguaje

Interface IService<TService, RService> {

execute(s: TService): Either<Error, RService>;

Nota: El aspecto debe recibir por inversión de dependencia respetando
el Principio de Inversión de Dependencias.

Logger de Performance

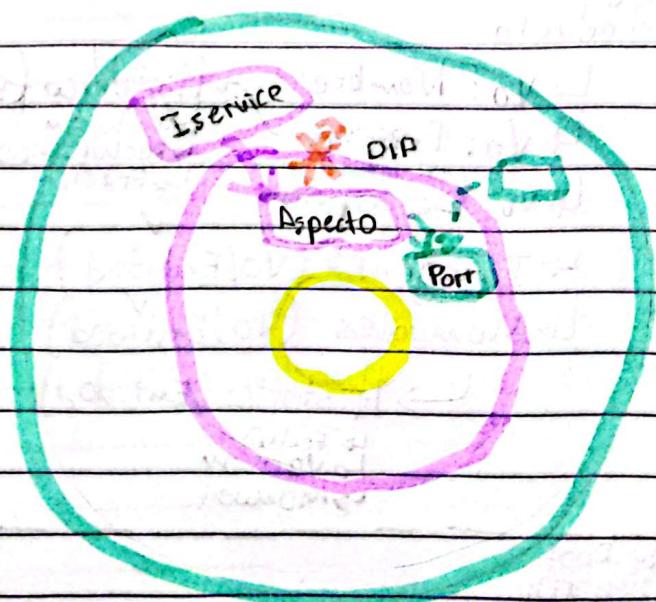
```
Class PerformanceServiceDecorator<TS, RS> implements IService<TS, RS> {
    -service: IService<TS, RS>; -logger: ILogger;
    execute(s: TS): Either<Error, RS> {
        let date = new Date();
        r = this.service.execute(s);
        let date2 = new Date(); → ?
        this.logger.save(date2 - date)
        return r
    }
}
```

Implementación - Composition Root

```
Let Service = ...  
new LoggingSD<(AsigTareaDTO, AsigTareaRepo)>(logger, ...);  
new PerformanceSD<(AsigTareaDTO, AsigTareaRepo)>(perfLogger,  
new AsignarTareaService(<-->(dayRepo, taskRepo)));
```

Hexagonal

Si algo de una capa requiere algo de otra, de adentro hacia afuera, entonces se utiliza una interfaz (DIP) para obtenerlo.



Clase #23 - Aspectos de los servicios de Dominio

Servicios de Dominio como Validaciones

