RESUMEN FINAL ALGO III

1. Buenas Prácticas en Programación
2. Mocking
3. Excepciones JUnit.
4. Ppios SOLID y Patrones de Diseño.
5. Diseño orientado a objetos
6. Metodología de desarrollo de software } no está, era mucha historia y muy dilógica
7. MVC - Modelo Vista Controlador
8. Java FX. → No esta.

---

(1) Buenas Prácticas en Programación

- Código de calidad y prácticas de XP

LEGIBILIDAD

↳ Código se ~~lee~~ escribe una vez y se lee muchas veces

↳ Usar: títulos de ≠ nivel
sangrías
signos de puntuación
tipografías especiales

1° NO REPETIR CÓDIGO

2° ESTANDARIZAR

↳ nombres
↳ formato (líneas en blanco, sangrías, etc.)
↳ Comentarios (código autodocumentado)

3°) NO sorprender
↳ suponer igual nivel del desarrollador que va a mantener mi código
↳ evitar lucirse
↳ No explicar lo obvio.

=> Nombres de Variables
- Descriptivos ⟶ distanciaEnMetros en vez que distancia
- No usar números ⟶ total1, total2 (X)
- términos del problema, no de la solución
  ↳ empleados en vez que arregloEmpleados
- Casos especiales
  - num, cant, long
  - max, min
  - i, k
  - tmp / temp, aux } temporales o auxiliares
- Evitar significados ocultos.
- Variables booleanas ⟶ no expresarlas en forma negativa
  ↳ poner esMujer ⟶ 1 sí / 0 no en vez que sexo ⟶ 1 mujer / 0 varón

=> Nombres de métodos
- Que describa todo lo que hace el método
- Que describa la intención y no la implementación

⇒ Métodos

↳ Tratar de que no dependan de una orden de ejecución.

↳ Recursividad sólo cuando haya legibilidad (sólo a nivel de 1 método)

↳ que afecte a la clase en la que está declarado

⇒ Parámetros

↳ no muchos (ninguno es lo mejor)

↳ ordenados con algún criterio (mantenerlo)

↳ chequear valores válidos a la entrada

↳ valores de entrada no deben cambiarse

⇒ Variables y métodos locales

↳ inicialización explícita

↳ 1 uso xa c/ variable

↳ vida de la variable lo más corta posible.

↳ usar métodos xa guardar partes de tests complic. locales

⇒ Comentarios

↳ Buenos xa aclarar código → aunque no deben abundar. (aunque deberíamos tratar de simplificar el código antes)

→ como resumen → aunque deberíamos tratar de separar en métodos

→ No aclarar redundancias

→

↳ antes del código al que se refieren

=> **Condicionales y ciclos**
- ↳ cuidar los >=, <=, >, <
- ↳ No más de 3 niveles
- ↳ en los "switch" => ordenar los case
- ↳ hacer obvia la manera de salir del ciclo

MEJORA de Desempeño
- mejorar el código sólo en las partes críticas.
- mejorar el hardware (analizarlo antes)
- Usar profilers

~~Todo~~

¿Cómo?

↳ Evitar el uso de memoria externa cuando se puede usar memoria interna

↳ En evaluaciones lógicas utilizan la opción "corto circuito"
- "(a and b) ifTrue: []" → no es necesario evaluar b si a es falso.
- "(a or b) ifTrue: []" → no es necesario evaluar b si a es verdadero.

↳ verificar si el entorno no lo hace sólo.

↳ Ordenar las preguntas en if compuestos y switch (evaluar primero las + frecuentes y menos costosas)

```
switch(mes){
  case 1,3,5,7,8,10,12 : dias=31; break;
  case 4,6,9,11: dias =30; break;
  case 2 if(anio.bisiesto()) dias =29;
        ~~dias=28~~
          else dias = 28;
          break;
  default : throw new MesInvalidoException();
}
```

↳ Calcular todo lo que se pueda antes de entrar a ciclos

→ int mejor que double
→ pasaje x valor siempre que se pueda
→ liberar memoria
→ lenguajes compilados o de menor nivel

Assembler >> C++ >> Java/ C#/ Smalltalk >> PHP/Python

⚠ Después de la mejora...
✓ Probar su efectividad

Construcción metodología incremental

XP => "Extreme Programming": Conjunto de prácticas de desarrollo centradas en la programación.
↳ llevan al extremo las buenas prácticas de la programación y cuestiones de sentido común.

- Probar ~~todo~~ el tiempo
- TDD
- Diseñar pruebas antes de codificar

} mucha disciplina

- Integración continua → evita que sea cada vez más complicado integrar código de varias fuentes.
- Minimizar documentación guardada si no se va a actualizar
- Revisar código todo el tiempo
- Pair-Programming → se mantiene más el foco
- Refactoring.

- fijan estándares pocos y estrictos
- micro-iteraciones
- diseñan pequeña porción, codificarla y probarla
- Preocuparse sólo x lo que se está haciendo (nada x adelantado)
- Simplicidad

PRIORIDADES de DOCUMENTACIÓN

1. Código autodocumentado
2. Comentarios
3. Pruebas unitarias
4. Pruebas automatizadas
5. UML
6. Docs. externos

} menor valoración aunque no se discuten

REFACTORIZACIÓN

↳ proceso de cambiar un sistema de software mejorando su estructura interna sin alterar el comportamiento externo del código.

↳ ¿Cuándo refactorizamos?

- agregamos una función
- arreglamos un error
- Code reviews

} regla de 3.

(2) MOCKING

↳ objetos que imitan el comportamiento de objetos reales de una forma controlada.

⚠ NO es lo mismo que el Patrón Proxy que permite proporcionar un sustituto para otro objeto. Un proxy controla el acceso al objeto original, permitiendo hacer algo antes o después de que la solicitud llegue al objeto original.

¿Para qué sirve?

↳ Pruebas de aislamiento con clases con dependencias

⇓

realizamos un "mock object" ya esas dependencias

¿Cómo?

↳ Con un framework Mockito. site.mockito.org

- crear el mock => Clase objetoNombre = mock(Clase.class);
- En la prueba:

```
when(objetoNombre.método1()).thenReturn(true);
when(objetoNombre.método2()).thenReturn(true);

{ usamos el mock con el assert

verify(objetoNombre, times(1)).método1();   ← cuántas veces queremos que lo haga
verify(objetoNombre, times(1)).método2();
```

=> podemos hacer que nunca medica cierto método
verify(objetoNombre, never()).metodo1();

También podemos crear nuestro propio objeto Mock donde tendremos que controlar la cantidad de veces que se utiliza un método de ese objeto y simularlo (ver ejemplo de Mocking Vehiculo y Conductor)

---

(3) Principios SOLID y patrones de Diseño.

PRINCIPIOS SOLID

1. SRP: Ppio. de responsabilidad única.
2. OCP: Ppio. de abierto/cerrado
3. LSP: Ppio. de sustitución de Liskov.
4. ISP: Ppio. de segregación de la interfaz.
5. DIP: Ppio. de inversión de dependencias.

1. Responsabilidad única.

=> Una clase debe tener una única razón xa cambiar. y ese cambio debe venir de un ~~refactor~~ requerimiento y no de un refactoring o arreglo de un bug.

=> Si vemos que un objeto tiene muchas responsabilidades, habría que crear más clases

2. Abierto / cerrado.

↳> Las clases deben estar abiertas xa la extensión y cerradas xa su modificación -

- Se debe poder cambiar el comportamiento s/ modificar el código ya existente.
- Se puede lograr utilizando herencia o delegación.

3. Sustitución de Liskov.

=> Clases heredadas deben poder ser utilizadas a través de su clase madre sin la necesidad de que el usuario sepa la diferencia.

- Se debe cumplir la condición "es un" al aplicar herencia.

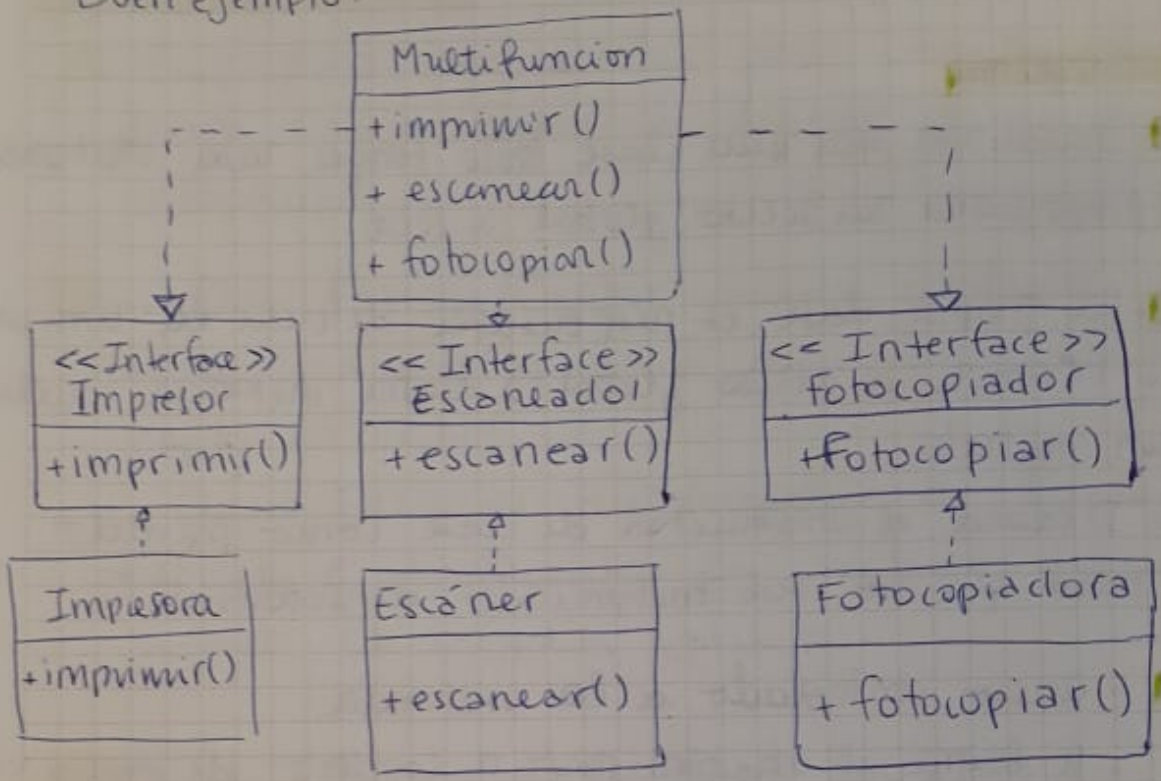
4. Segregación de la interfaz.

=> Los clientes no deben ser forzados a ~~utilizar~~ depender de métodos que no utilizan.

- Muchas interfaces >>> interfaz de propósito específicos general.

- Necesario aplicarlo cuando se tiene una clase con varios métodos (de los cuales solo me interesan algunos)

Buen ejemplo.



5. Inversión de dependencia

=> Se debe depender de las abstracciones y no de las implementaciones.

PATRONES de DISEÑO → c/patrón tiene { propósito, solución, consecuencias }

Clasificación según su intención:

(A) Singleton

P: Garantiza que una clase tenga una instancia y proporciona un acceso global a ella.

S:
- La propia clase es responsable de crear la única inst.
- Permite el acceso global a dicha instancia mediante un método.
- Declara el constructor de clase como privado para que no sea instanciado directamente.

C:
- Acceso controlado a la instancia
- Se logra el objetivo pero a cambio de ensuciar la clase
- Se lo considera un patrón fácil pero intrusivo.

(B) Multiton (2..n)

P: Garantiza que una clase sólo tenga varias instancias conocidas, y proporciona un pto. de acceso global a ellas.

S: Se implementa igual al singleton pero con un mapa en vez de con un atributo. El método getInstance recibe el nombre de la instancia.

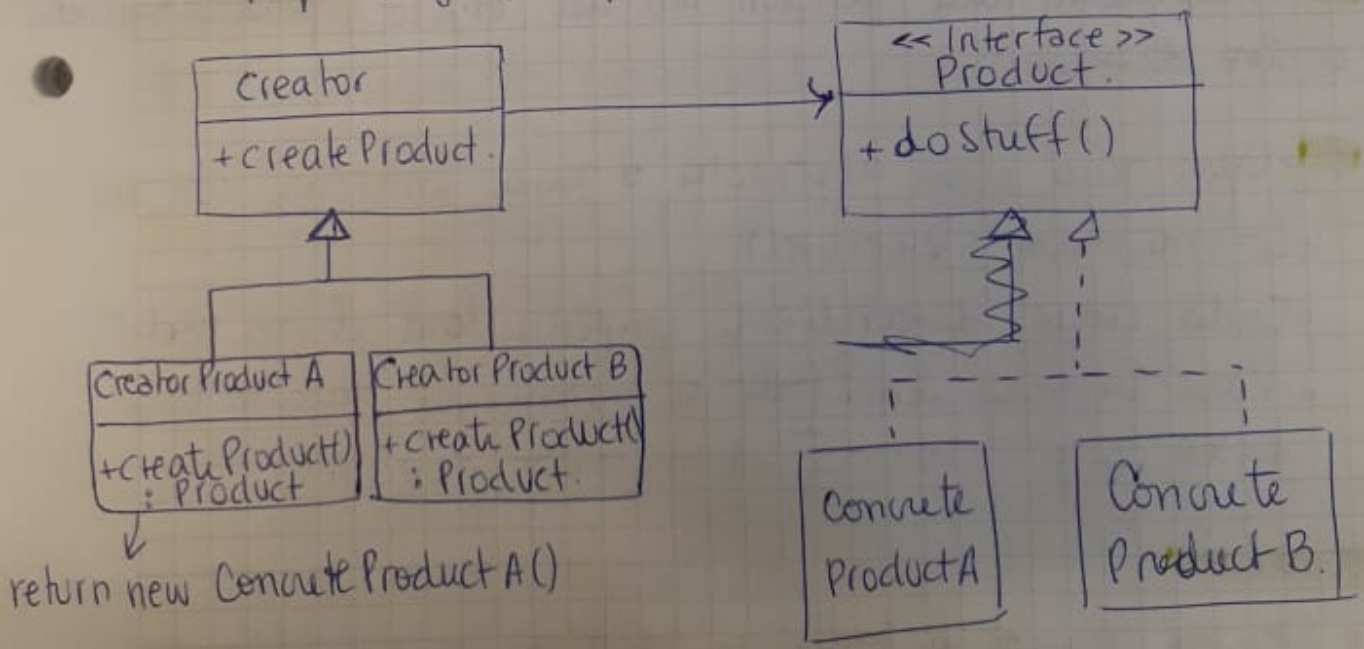
Aplicabilidad: Application loggers (log de producción o de Debug).

(c) Factory Method

P: Proporciona una interfaz para crear un objeto, delega en sus hijos la decisión de qué objeto instancian.

S: • Clase Factory/Creator que sea abstracta que marca una interfaz común para la creación del objeto en cuestión.
• Clases Factories que hereden de Factory abstracta e implementen EL método definido cuando la instancia concreta.

C: • Independencia de las clases concretas
• Permite intercambiar el objeto creado de manera rápida y transparente.




```
Product p = createProduct()
p.doStuff()
```

(D) Abstract Factory

P: Proporciona una interfaz x a crear familias de objetos relacionados que dependen e/ sí, sin especificar sus clases concretas.

S: • Crear Clase Factory abstracta que provea una interfaz común de creación de las familias de objetos.
• Crear clases Factories que hereden de la clase Factory y e implementen los métodos definidos creando las instancias concretas.

C: • Independencia de las clases concretas.
• Permite intercambio de familias de objetos de manera rápida y transparente.

(E) Command

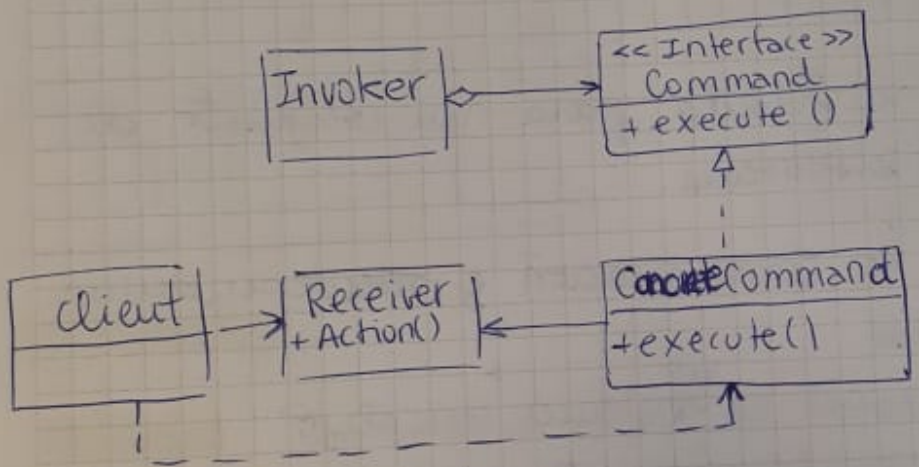
P: Encapsular una petición en un objeto, parametrizando a los clientes

S: • Crear una clase abstracta o una interfaz con un sólo método execute().
• Cada clase derandiente implementará el método.
• Para invocar el método se instanciará una de las clases, y se invocará al método execute().

Motivación:

• Objetos como botones y menús que realizan una petición en respuesta a una entrada de usuario
• Transacciones.

C: • Permite mantener referencias a métodos.
• Desacopla el objeto que invoca la operación de aquel que sabe cómo realizarla.
• Órdenes → objetos de 1ra clase.
• Se pueden hacer órdenes compuestas.
• Es fácil añadir nuevas ordenes, ya que no hay que cambiar las clases existentes.



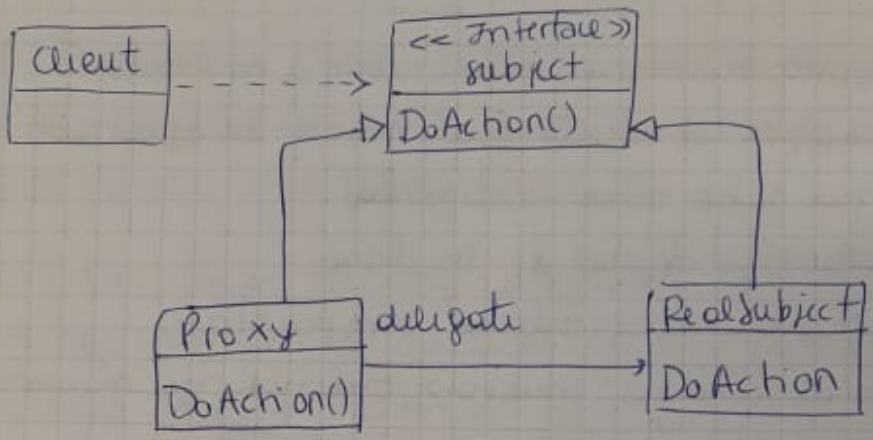
(F) Proxy.

P: Proporciona un representante o sustituto de otro objeto para controlar el acceso a este.

S: • Crear jerarquía en la que intervengan el objeto original y el objeto proxy.
• En el objeto Proxy habrá una referencia al obj original
• Se redirigen todas las llamadas al Proxy antes de acceder al obj. original.

M: Retrasar el coste de creación e inicialización hasta que realmente sea necesario (archivos con imágenes)

C: • agregar funcionalidad de forma transparente a la app.



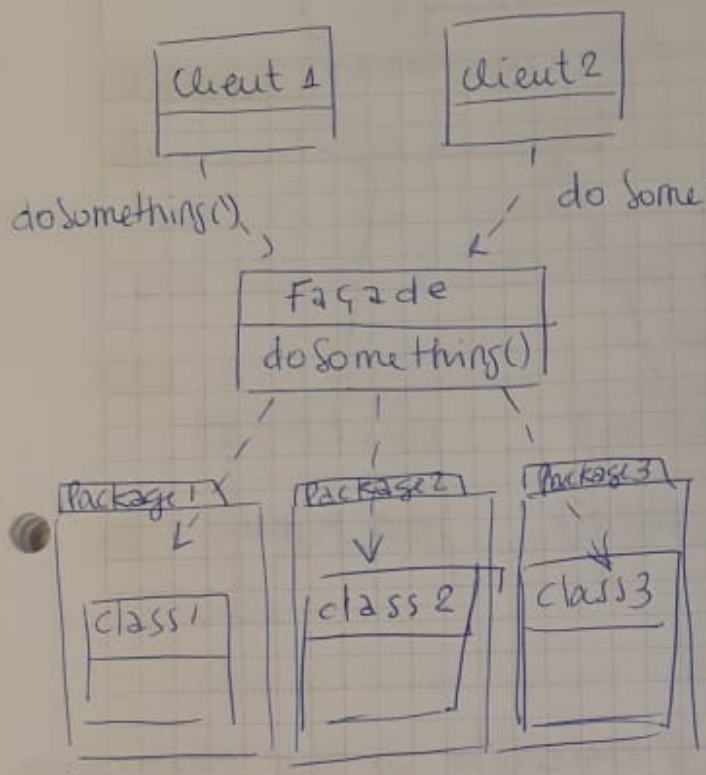
(G) Facade.

P: Interfaz unificada para un conjunto de interfaces de un subsistema.

M: Minimizar la comunicación y dependencia entre subsistemas.

A: • Interfaz simple x a subsistema complejo
• Muchas dependencias entre clientes y clases que implementan una abstracción.
• Dividir en capas nuestros subsistemas.

Beneficios:
Oculta a los clientes los componentes del subsistema
Promueve un débil acoplamiento e/ el subsistema y sus clientes
No impide que las Apps usen las clases del subsistema en caso de que sea necesario.



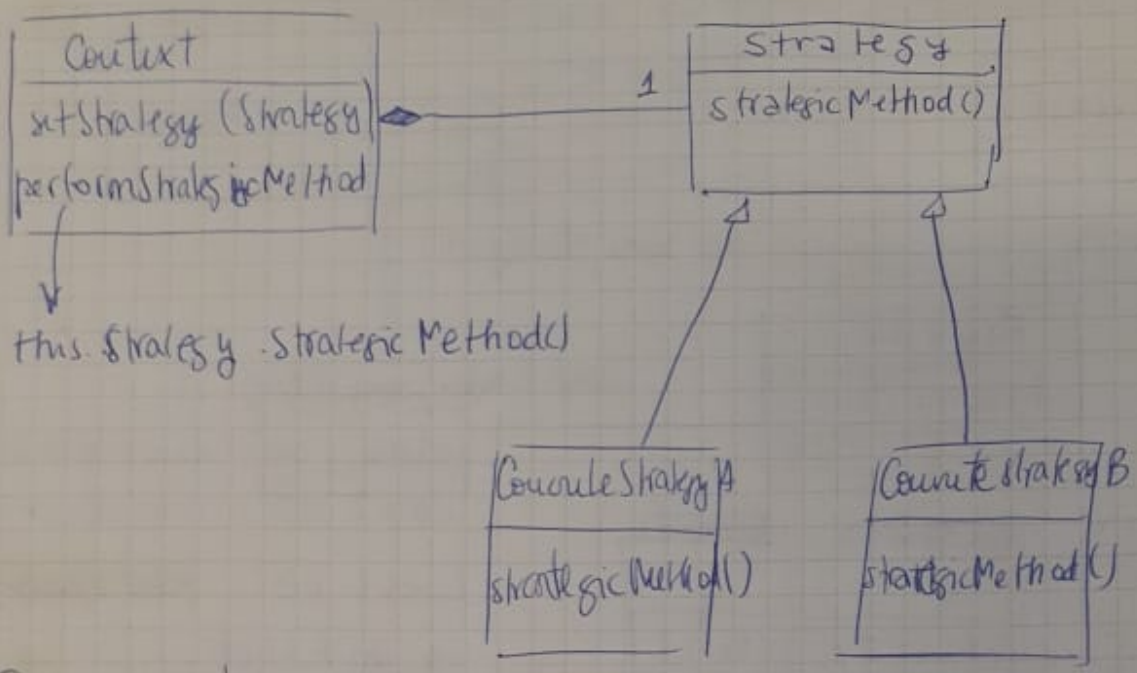

```
doSomething(){
  class1 c1 = new class1();
  class2 c2 = new class2();
Class3 c3 = new Class3();
c1.doStuff();
c3.setX(c1.getX());
return c3.getY();
}
```

(H) Strategy

Necesidad: Un mismo objeto debe ~~ser determinado~~ poder tener un comportamiento que debe ser determinado en tiempo de ejecución.

Solución: • Delegar comportamiento.
• Armar jerarquía c/ los (7) comportamientos
• Inyectar el comportamiento al objeto a través de un método o de un constructor.

Consecuencias: Se eliminan condicionales
Se crea jerarquía // a la jerarquía base.
Comportamientos agrupados x familias
A veces no es fácil aislar comport.
Ahora no alcanza



(I) Template.

P: Define en una operación el esqueleto de un algoritmo delegando en las subclases algunos pasos y les permite redefinirlos sin cambiar la estructura del algoritmo.

A: Para que las subclases redefinan el comportamiento que puede variar.

B: Reutilización de código.
Extraen comportamiento común de las clases de la biblioteca

(J) State

P: Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecerá que cambia la clase del objeto.

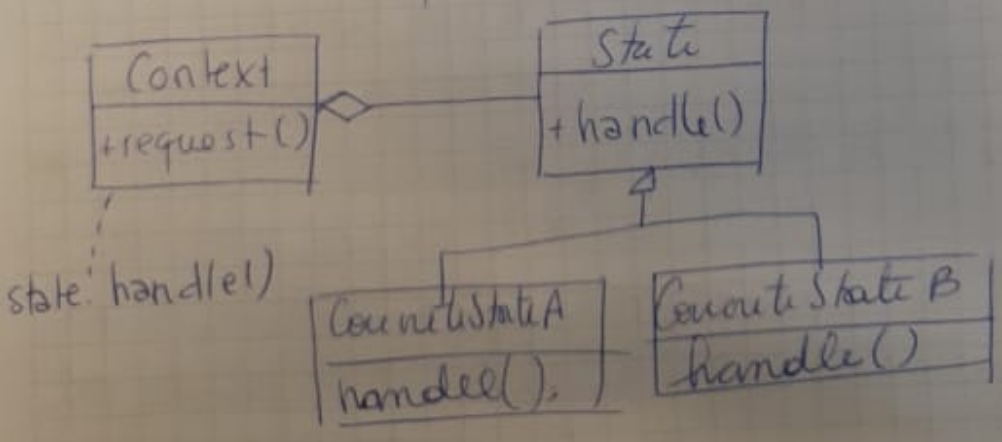
M: TCP —3 estados→ establecida, escuchando, cerrada

A: El comportamiento de un objeto depende de su estado

Operaciones tienen largas sentencias condicionales con múltiples ramas que dependen del estado del objeto. Estado representado x 1 o + CTES. enumeradas.

B: Localiza comportamiento dependiendo del estado y divide dicho comportamiento en (7) estados.

Transiciones el estados explícitas



(k) Decorator.

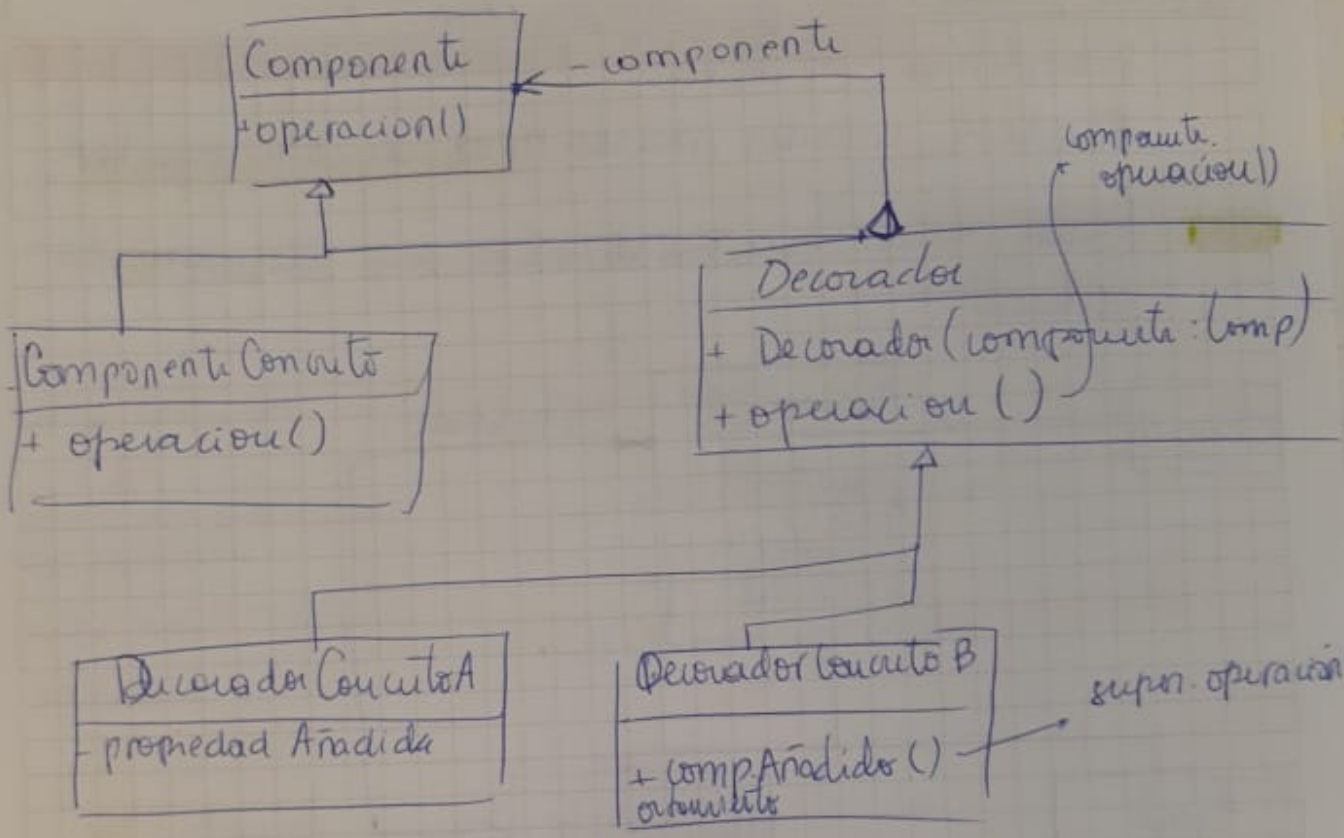
P: Asigna responsabilidades adicionales a un objeto proporcionando una alternativa flexible a la herencia xa extender su funcionalidad

M: Interfaces de usuarios a las que se agregan propiedades o comportamientos.
Cambio de piel.

A: Añadir objetos individuales de forma dinámica y transparente, es decir, sin afectar a otros objetos.
Responsabilidades que pueden ser retiradas
Cuando la extensión mediante herencia no es viable.

B: + flexibilidad que la herencia estática
Evitar clases cargadas de funciones en la parte de arriba de la jerarquía.

Desventajas
Un decorador y su componente no son idénticos
Muchos objetos pequeños.



(L) Composite.

P. Compone objetos en estructuras de árbol ya representan jerarquías de parte-todo

M. Aplicaciones gráficas como los editores de dibujo y los sistemas de diseño permiten agrupar comp. simples en más grandes.

Manejo de excepciones.

A. Quiere representar jerarquías de objetos parte-todo

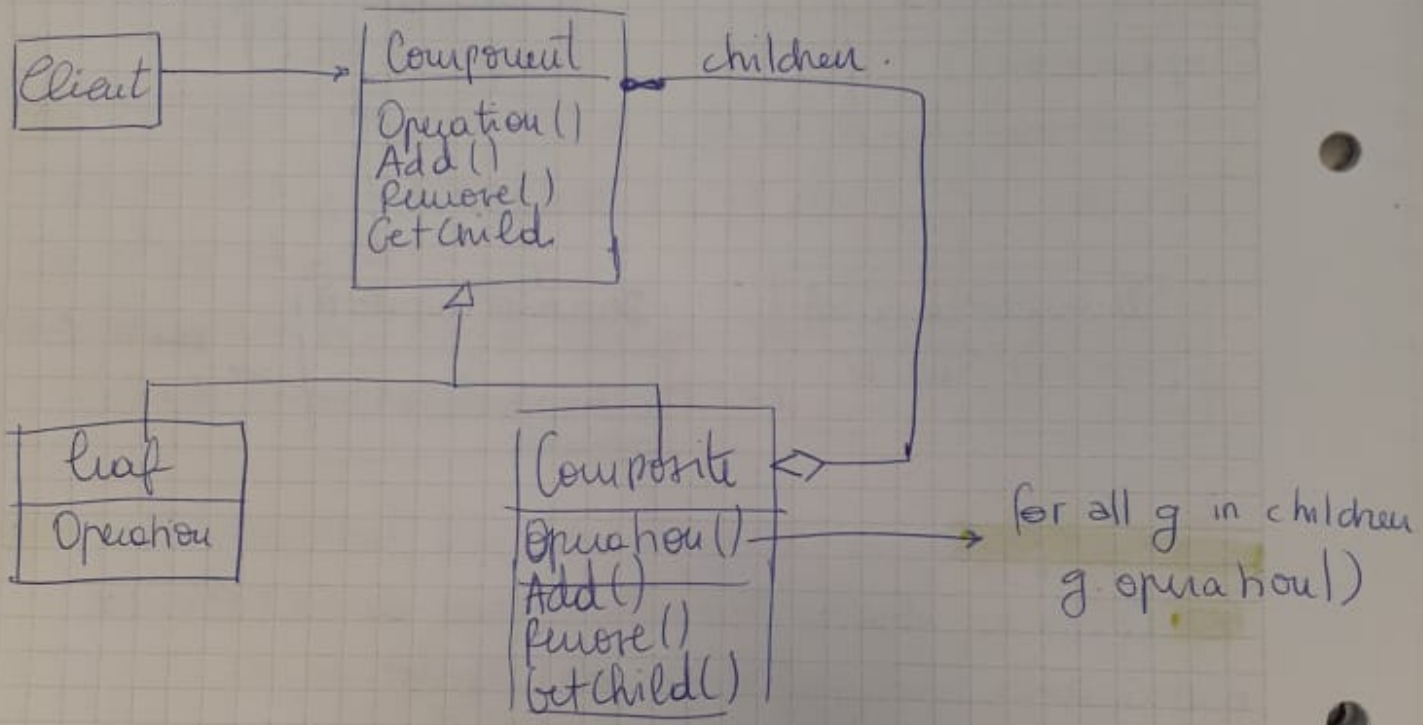
Quiere que los clientes sean capaces de obviar las diferencias entre composiciones de objetos y los objetos individuales. Los clientes tratarán a todos los objetos de manera uniforme.

B: Definir jerarquías de clases formadas x objetos primitivos y compuestos.

Simplifica el cliente.

Facilita añadir nuevos tipos de componentes

D: Puede hacer que mi diseño sea demasiado general.



5. Diseño Orientado a Objetos

- **MODULARIZACIÓN**

✓ Cohesión y acoplamiento → comprensibles, mantenibles, utilizables

alta cohesión ⇓
=> coherencia intrínseca del módulo
=> respecto del módulo solamente

bajo acoplamiento ⇓
=> relación con otros módulos
=> unión externa

✓ Cohesión y acoplamiento en clases
- Representa una sola entidad
- Pocas dependencias c/ otras clases

✓ Cohesión y acoplamiento en paquetes
- Poca dependencia e/ paquetes
- Deben distribuir una unión global del sistema

✓ ~~Cohesión~~ " en métodos
- Un solo tabo activo
- Pocos parámetros & pocas llamadas a otros métodos
- Parámetros x valor en lo posible.

- Patrón Observer

Intención: Definir dependencia uno-a-muchos e/ objetos de manera que cuando un objeto cambia su estado, todas sus dependencias sean notificadas y actualizadas automáticamente.

Motivación: Mantener la consistencia e/ objetos que dependen entre sí, reduciendo el acoplamiento y maximizando la reusabilidad.

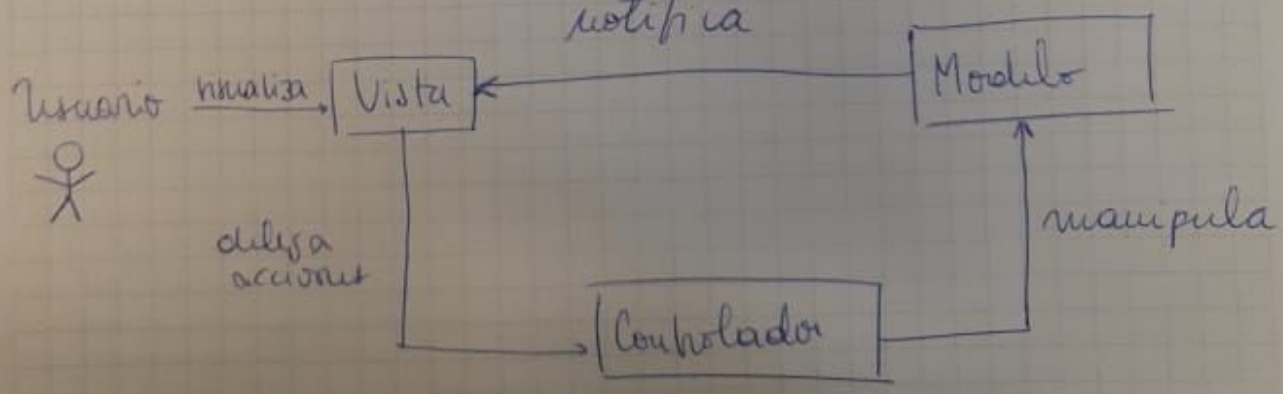
- Patrón Strategy => se puede pensar el controlador como una instancia del strategy.

- **VISIBILIDAD**
  - → lo + privado que se pueda
  - → garantiza que se pueda modificar código afectando al mínimo posible de clientes

- **MVC: Modelo Vista Controlador**
  - → Patrón de arquitectura diseñado xa la construcción de interfaces de usuario.
  - → Separación de responsabilidades

3 objetos

1. **Modelo**: lógica y entidades de negocio o dominio de nuestra aplicación
2. **Vista**: formas en que los objetos del modelo se muestran y representan al usuario.
3. **Controlador**: definen cómo la interfaz de usuario reacciona a las acciones del usuario.



- DISEÑO de CLASES.

Razones: Modular una entidad del dominio
Modular una entidad abstracta
Aislar complejidad y detalles de implementación
Separar código poco claro en clases especiales
Agrupar operaciones relacionadas en una clase de servicios

- DELEGACIÓN SOBRE HERENCIA.
  - Herencia es estática
  - Delegación otorga mayor flexibilidad
  - Herencia: cuando se va a reutilizar la interfaz.
  - Delegación: cuando se van a reutilizar algunas responsabilidades

- POLIMORFISMO.
  - ↳ Evitar condicionales.

- DISEÑO de PAQUETES
  - ↳ bajo acoplamiento
  - ↳ Construir las agrupaciones de clases de modo tal que sí van a reutilizarse
    - ↳ las clases que se cree que se van a usar conjuntamente deberían ubicarse en el mismo paquete.
    - ↳ clases que cambian poco separadas de aquellas que cambian mucho