08/09/2016

ACTIVIDAD 2

Grupo	Nombre	Apellido
1	Felipe	Riquelme.
1	JAVIER	DIAZ
1	JOSE MARIA	DE LA TORRE
	Gerardo ~~Olmos~~	Olmos
2	Benjamin ~~Ibarre~~	Ibarre
	María Fernanda	Sepúlveda
3	Antonia ~~Fontaine~~	~~Fontaine~~
3	Agustin ~~Gomez~~	Gomez
3	Diego	Sinay
4	Baltazar Ochagavia	BOB
4	Hector Quiroga	HQC
4	Carlos Aguirre	CAO
5	Cristóbal ~~Martinez~~	Martinez
5	Diego	Passi

Actividad clase 8/09/2016 Grupo 2

1. Singleton: Dado que el registro es manual, tener una sola instancia asegura consistencia

Aún así con singleton no nos ayuda a controlar permisos.
Puede que el validador sea único, pero hay que modificar una brutalidad de código para validar cada endpoint

2. Builder: El mensaje tiene distintas representaciones (audio, mstyrmail)

Builder sirve, pero no por si solo en este caso donde hay distintos tipos excluyentes

3. Prototype: Utiliza una misma plantilla y dependiendo de la operación decide como llenarla.

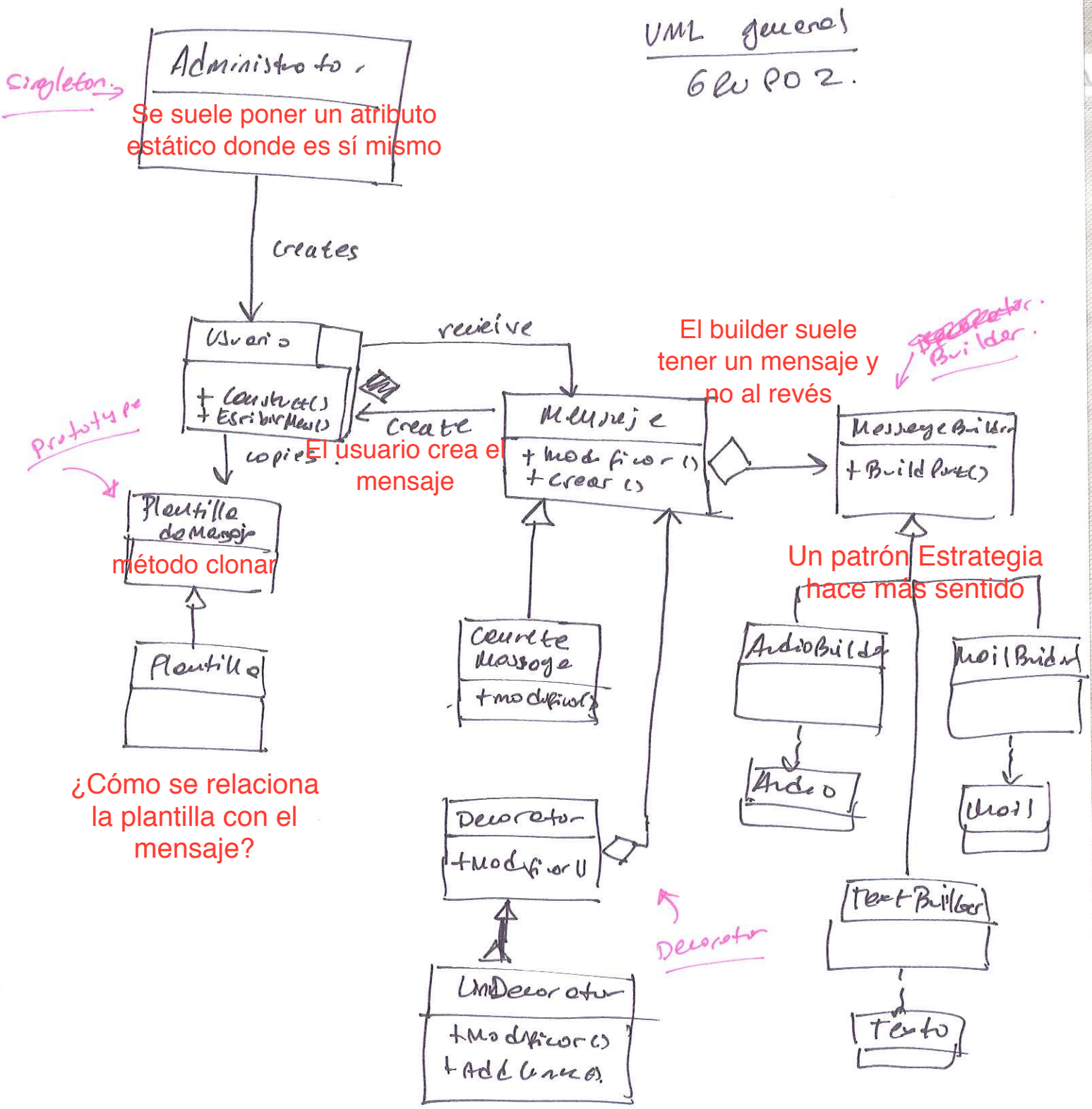
4. Decorator: Permite cambiar dinamicamente el comportamiento

5. Adapter: Implementa distintas interfaces para los distintos proveedores y los adapte para el consumo del cliente

6. No aplica :)

7. Chain of resp: Valida secuencialmente los mensajes.

UML general
GRUPO 2.

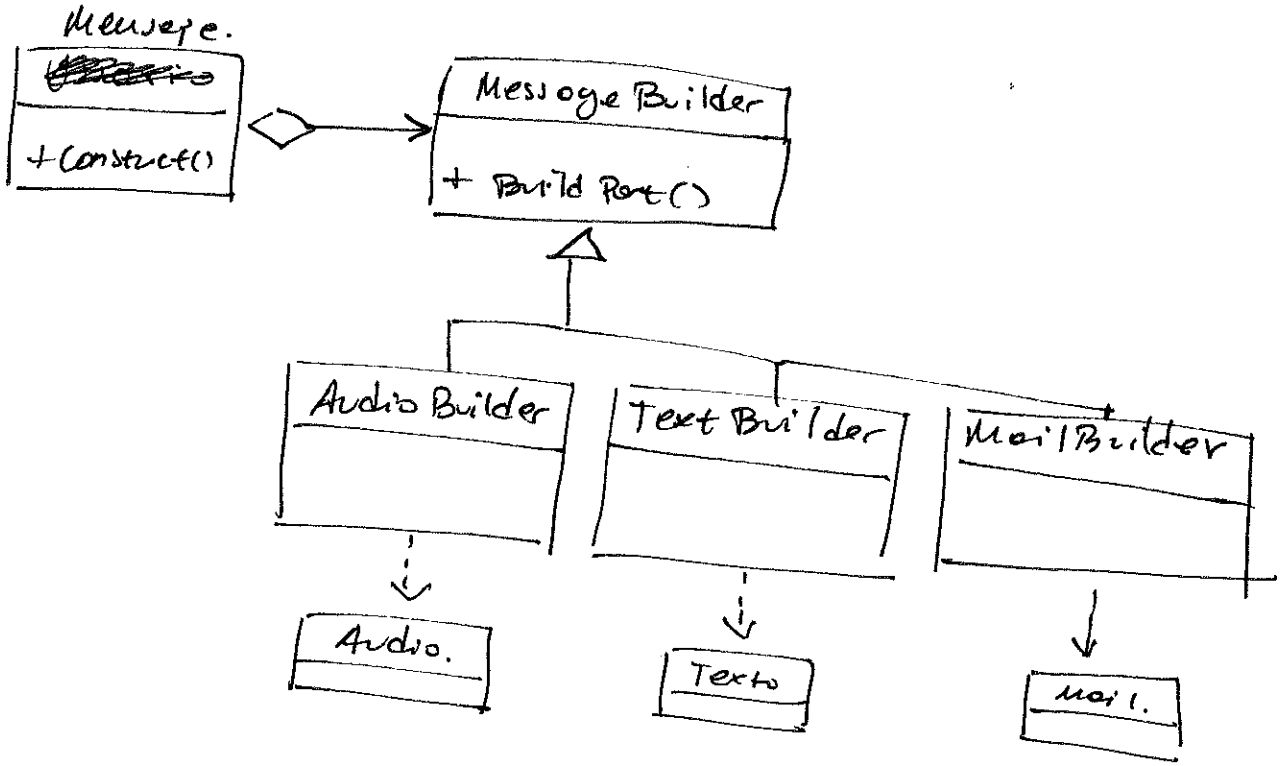


Diagramas UML.

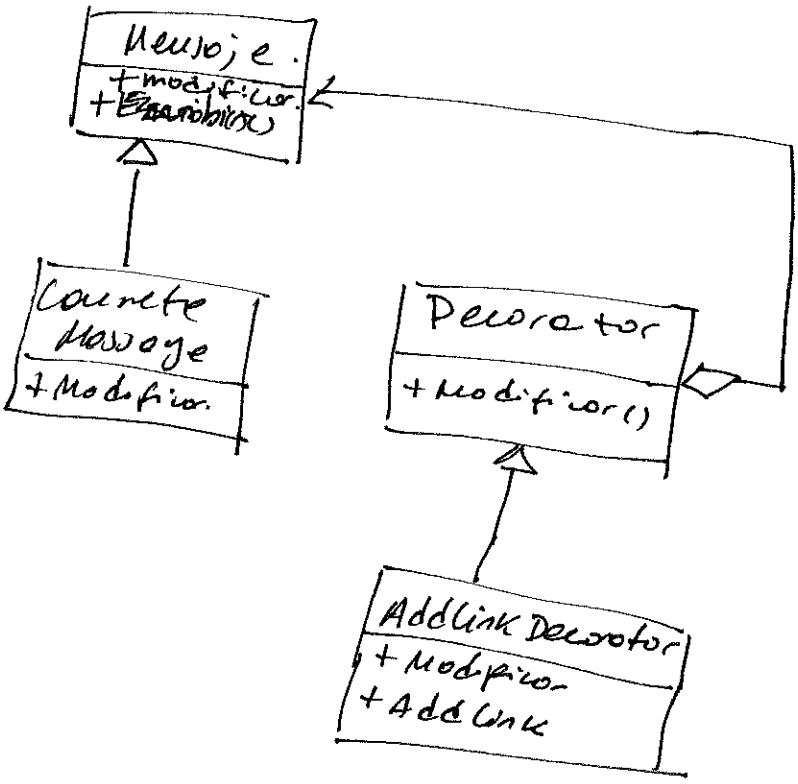
UML individual

GRUPO 2

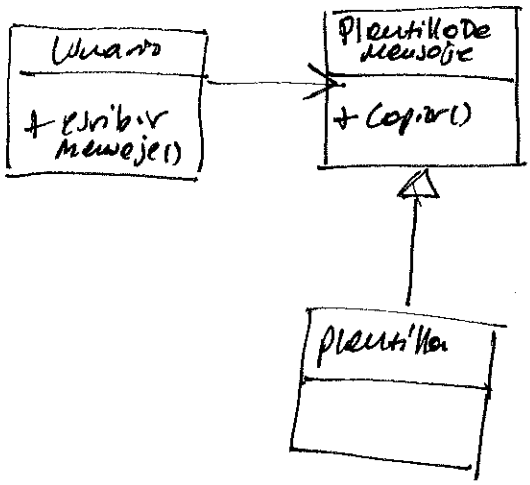
2. Builder: Construye inst. complejos.



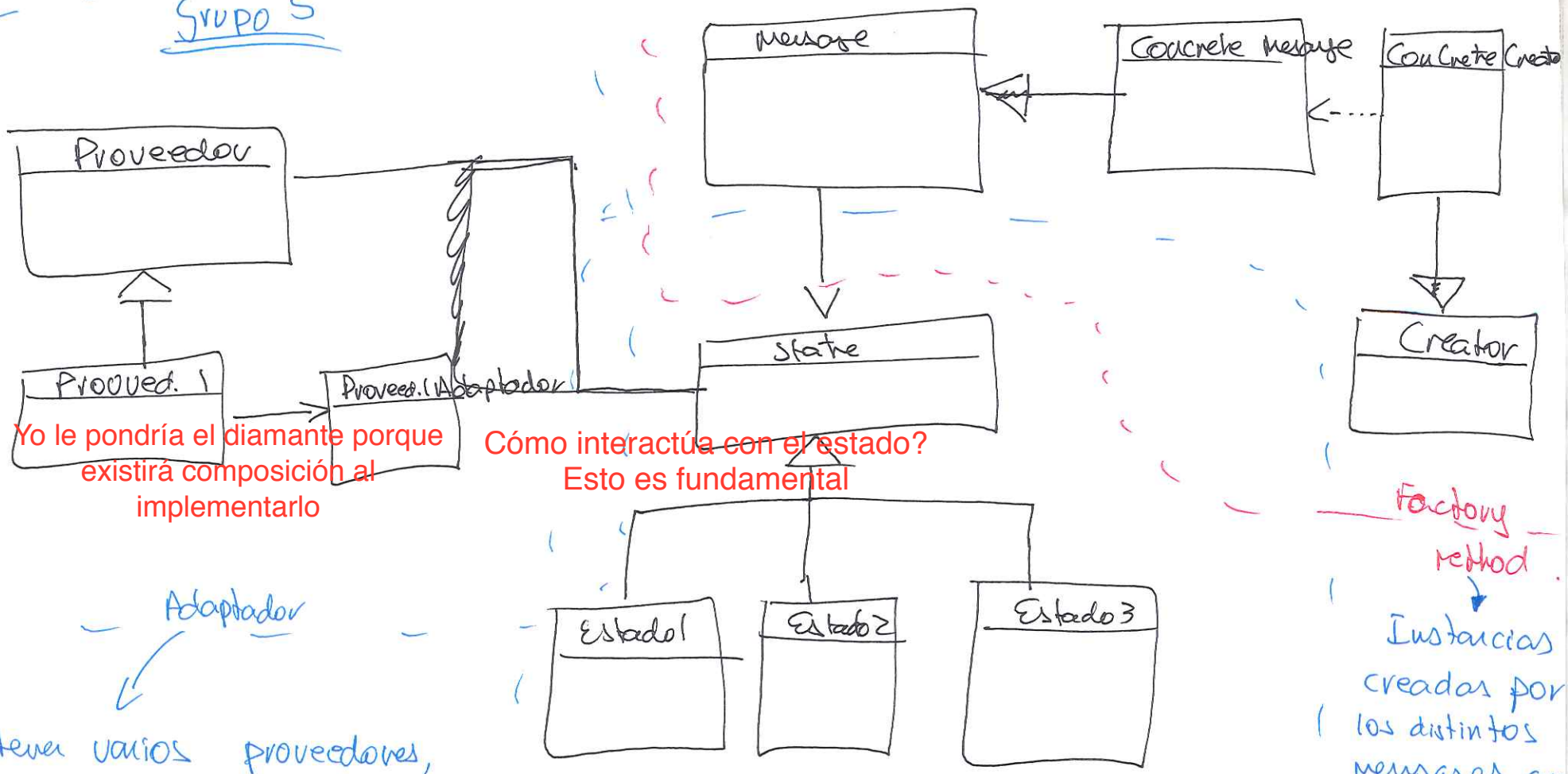
4. Decorator



3. Prototype


Usuario
+escribir mensaje()
PlantillaDe mensaje
+Copiar()
plantilla

Grupo 5



Yo le pondría el diamante porque existirá composición al implementarlo

Cómo interactúa con el estado? Esto es fundamental

Factory method

Instancias creadas por los distintos mensajes que te pueden enviar.

Adaptador

Al tener varios proveedores, dado un mensaje que envía el state, te manda el mensaje a ese proveedor segun el formato que él necesite.

+ estrategia

State

Según el estado de un mensaje, su bugo y cantidad de gente a enviar, elige que proveedor utilizar.

Me parece bien

Patrones

Grupo 5

1.- Se podría usar un patrón Chain of Responsability para ver que un usuario esté registrado. La razón de esto es que se pueden agregar luego, más validaciones y no se quedará atado a solo que esté registrado. (P.E. que esté activado).

Su solución con respecto a la que habíamos considerado (proxy) es una generalización y es válida :)

2.- Se podría usar Factory Method para el Req. 3 ya que se tienen ~~tiene~~ varios templates (pasos de creación) y se podrían encapsular, luego como siempre se obtiene un mensaje

3.- Se podría usar Prototype porque se tiene un objeto base (mensaje) y varios tipos de mensajes definidos (texto, correo, etc) de como transmitirse.

4.- Se podría usar Decorator para el requisito 4 ya que se puede agregar el comportamiento que se necesite para el mensaje, encapsulando estos comportamientos en decoradores.

5.- Se podría utilizar Adaptador para la comunicación con los proveedores (Req. 5) ya que las interfaces de cada uno son distintas, y el envío de mensajes no debería depender de esas interfaces.

6.- Para elegir el comportamiento de envío de mensaje según las características del mensaje (Req. 6) se podría utilizar el patrón State para elegir que proveedor usar según el estado del mensaje.

7: Para el requisito 7 se podría usar el patrón Fly-weight ya que al tener muchos mensajes se debe lidear con muchos objetos. No ataca al problema de la secuencia de operaciones a realizar

1: Factory: para crear usuarios. Habría una clase UserCreator (concrete creator) que ocuparía el Admin para crear usuarios (quienes a su vez podrían ser Admin), por lo que sería el "concrete product". El sistema se beneficiaría ya que la lógica de crear usuarios nuevos estaría fuera de la clase Admin (Alta cohesión).

Pero no ayuda a controlar permisos

2. Builder: para crear distintos tipos de mensajes ya que todos tienen un proceso prácticamente igual de construcción (todos tienen una forma de nombre, creador, recipiente etc). Builder sirve, pero en este caso no por si solo

3. Prototype: para crear las plantillas. Como las plantillas cumplen el propósito de dar una estructura común y predefinida para cada tipo de mensaje, tendríamos una clase "concrete prototype" para cada plantilla según el tipo de mensaje.

→

4. Decorator: para agregar funcionalidad dinámicamente a los mensajes. Lo elegimos ya que permite agregar la flexibilidad requerida sin modificar la clase de cada mensaje para agregar funciones.

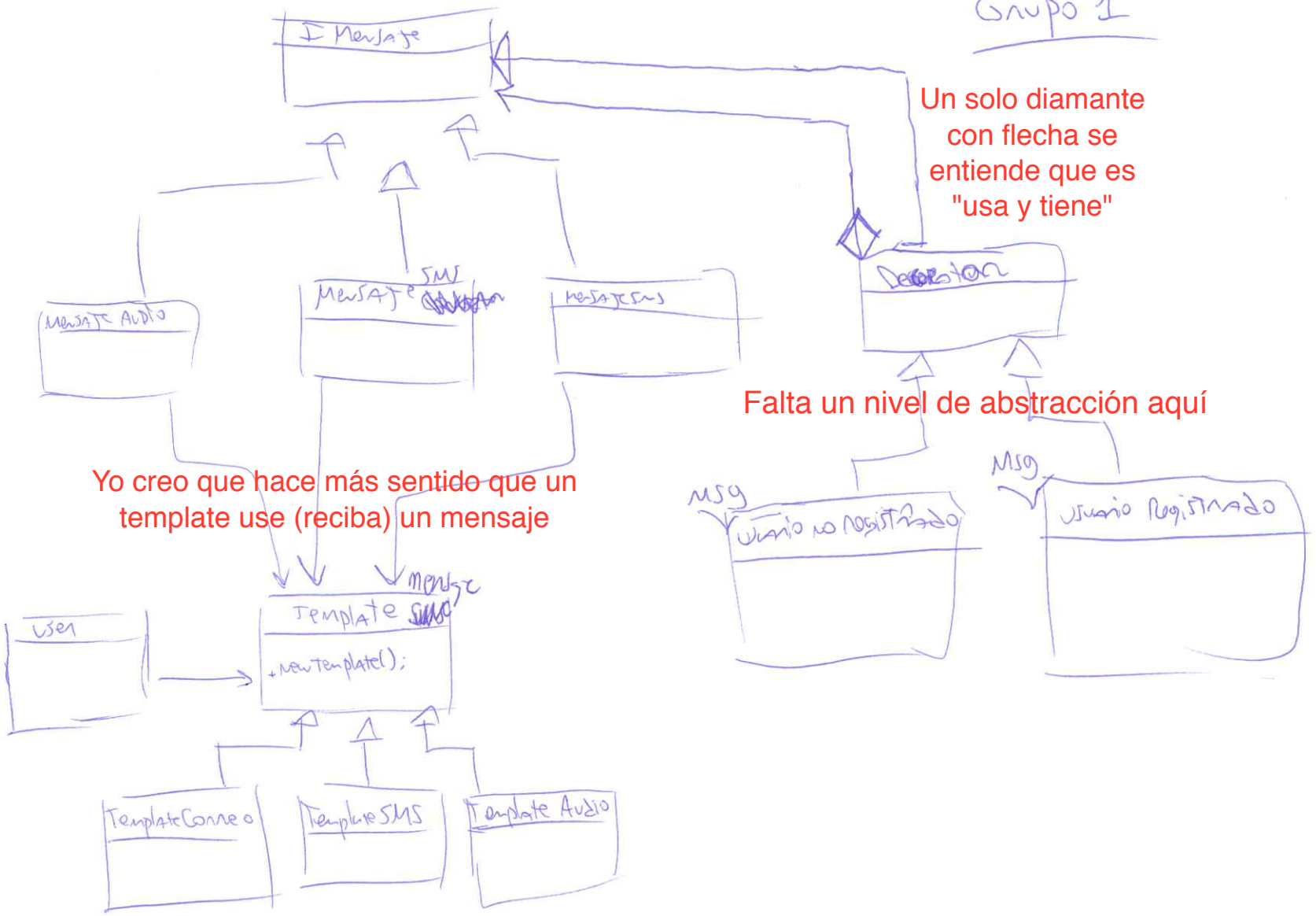
6. Chain of Responsibility: para elegir qué proveedor externo ejecutará el envío de mensajes. La idea es tener handlers que correspondan a cada proveedor y seguir la lógica de negocios elegir qué handler ocupar. De esta manera disminuimos el acoplamiento a un proveedor específico y podemos decidir cual ocupar dinámicamente.

Es válido, el único problema es que sería un flujo "greedy" y lo tomaría el primero que se considere apto, pero no necesariamente el mejor

5. Adapter: para manejar la posibilidad de cambio de proveedores. La interfaz para comunicarse con los proveedores se deja en una clase adaptador para evitar diseñar el sistema en torno a un proveedor específico (y su interfaz específica).

7. Observer: para que quien envíe el mensaje pueda observar a los receptores y enterarse de si recibieron o no los mensajes.

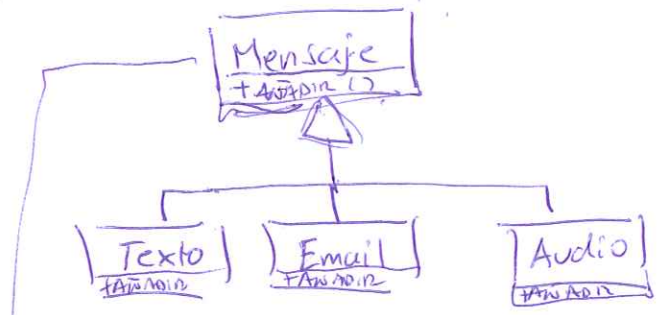
Grupo 1



Al momento de implementar van a existir problemas de Open/closed principle y de Sustitución de Liskov

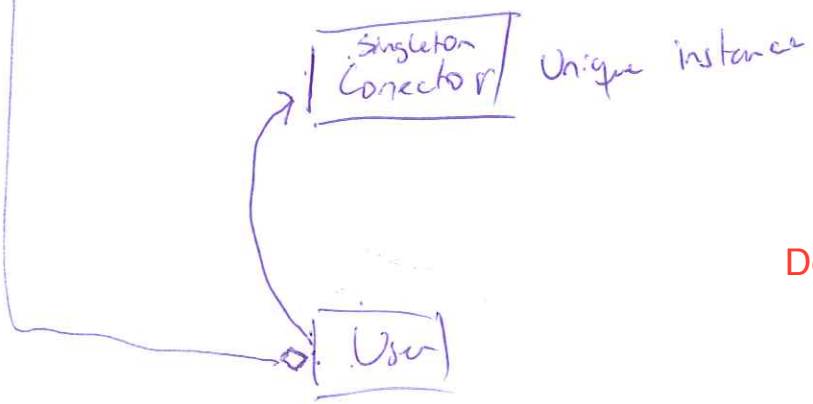
~~Factory~~

FACTORY



- Abstract Factory
- Singleton.
- ~~Builder~~
- DECORATOR

No hay diagrama de una factory, solo se aprecia herencia



Decorador??

~~MODIFICACIÓN~~ FACTORY: SE ENCUENTRA A LA HORA DE CREAR UN MENSAJE. EXISTEN 3 CLASES QUE HEREDAN DE MENSAJE (TEXTO, EMAIL Y AUDIO), PERO LA CLASE MENSAJE DEVUELVE EL ~~TIPO~~ DE UN MENSAJE.

DECORATOR: ~~TODO~~ CADA MENSAJE ES GENÉRICO PERO SE PUEDEN AÑADIR COSAS DINAMICAMENTE. POR ESO EL METODO AÑADIR.

SINGLETON: Funciona como un Load Balancer, que obtiene distintas fuentes de información desde distintos canales, y la transmite de manera secuencial por un solo canal

Los singletons son bastante cuestionados hoy en día. No se justifica su uso aquí

Baltazar Ochagavia
Carlos Aguirre
Hector Quiroga

G4

Abstract.

a)

1: Factory ~~method~~: asociado a la funcionalidad 2, donde el factory define una interfaz, pero cada mensaje se define que clase instanciar (texto, email o audio).

2: Adapter: asociado a la f. 5, ya que se necesita que las clases interactuen, por lo que debe hacerse que las interfaces sean compatibles

Builder sirve, pero no lo están entendiendo bien

3- Builder: asociado a la f. 3, ya que hay una creación genérica pero con ciertas variaciones o representaciones distintas.

4- Decorator: asociado a la f4, ya que agrega responsabilidad adicional. En este caso agregar información adicional al mensaje.

5 - Facade: asociado a la f6. ya que define un interfaz (reglas) y ~~garantiza~~ [illegible] más económico.

Fachada es muy simplista para este caso, al final solo nos sirve para abstraernos un poco, pero todavía dentro de este todavía habría un caos

6: Singleton: asociado a la f. 7, ya que hace que haber 1 sola entidad que regule el envio de mensajes. Igual que un Load Balancer. con acceso global.

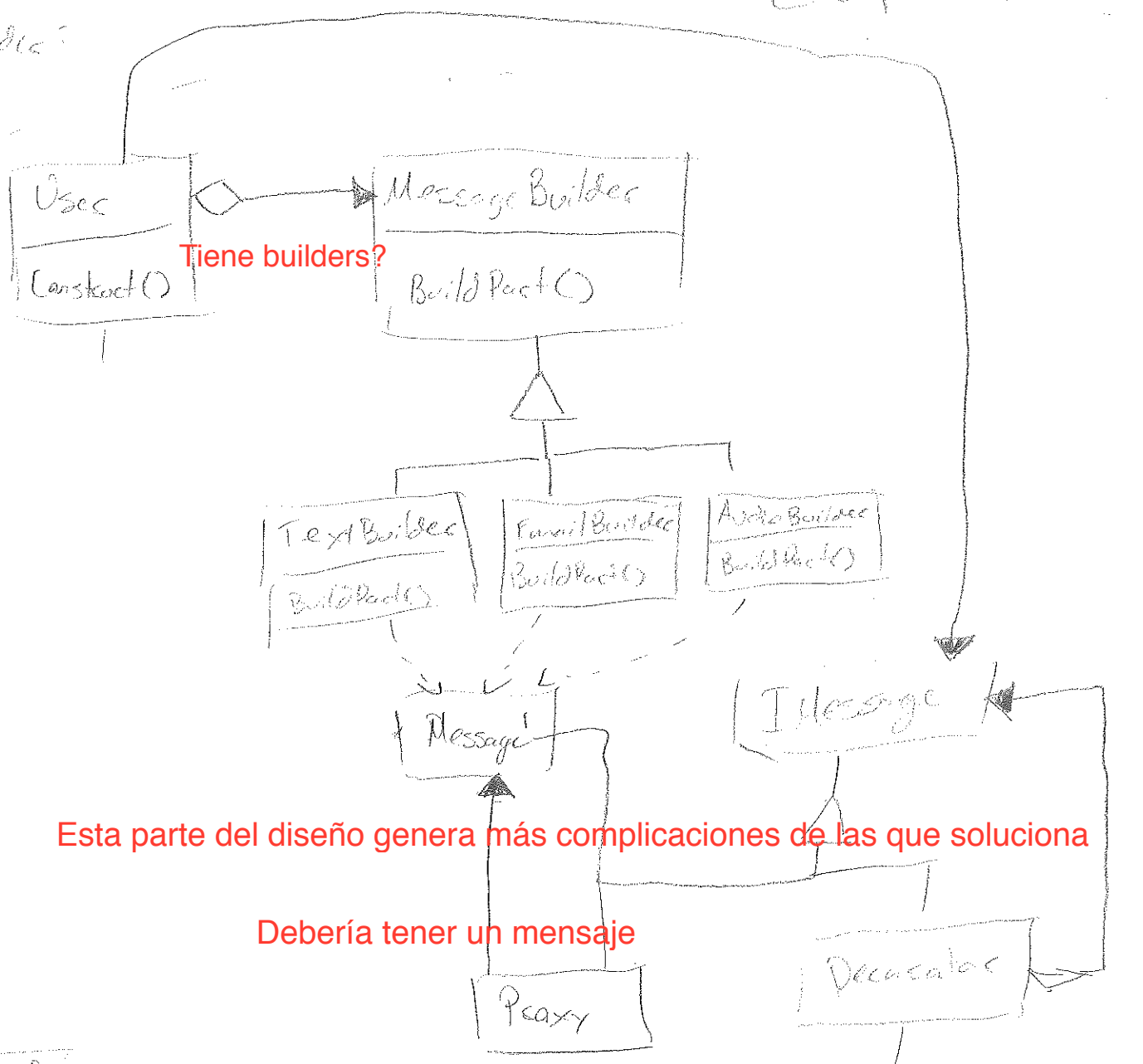
No nos garantiza la funcionalidad esperada

7- Prototype: asociado a la f3, ya que el administrador crea instancias de los usuarios manualmente, copiando de otros usuarios

7- Prototyp: f3: ya que los mensajes (en broadcast) tienen la misma estructura y formato

Grupo 3

b.) Builder:

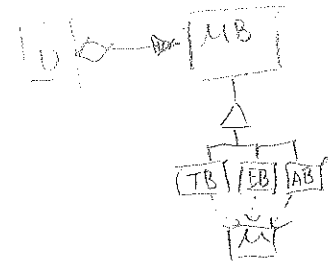


Tiene builders?

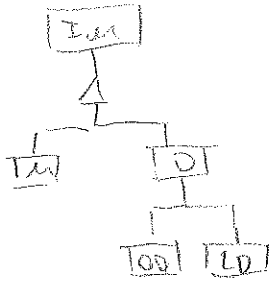
Esta parte del diseño genera más complicaciones de las que soluciona

Debería tener un mensaje

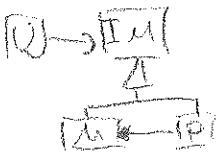
Builder:



Decorator:



Proxy:



Actividad 2

Grupo 3

Ojo que los tipos son excluyentes, un único builder genera problemas de abstracción

a.) Builder: en el requisito 2 se deberían crear 3 tipos de objetos complejos similares y se podría aprovechar el mismo proceso de construcción

Decorador: se recibe un mensaje y se le va agregando comportamiento de manera dinámica (requisito 4)

Prototipo: en el requisito 3 se podría utilizar este patrón, ya que hay una cantidad limitada de colores, los objetos son similares y las clases a crear se especifican en tiempo de ejecución

Adaptador: en el requisito 5 se podría utilizar el adaptador para transformar los distintos formatos de los distintos proveedores a algo con lo cual el sistema

Mediador: en el requisito 6 se podría utilizar este patrón ya que se debe mejorar la comunicación con todos los proveedores (algoritmo) y escoger el más económico

Ya sí, puede ser

Observer: en el requisito 3 se podrían utilizar observers para notificar al emisor que los mensajes han sido recibidos por los receptores

proxy: en el requisito 1, el sistema crea un proxy del emisor y el proxy se encarga de enviarle al receptor y avisarle que el mensaje fue válido.