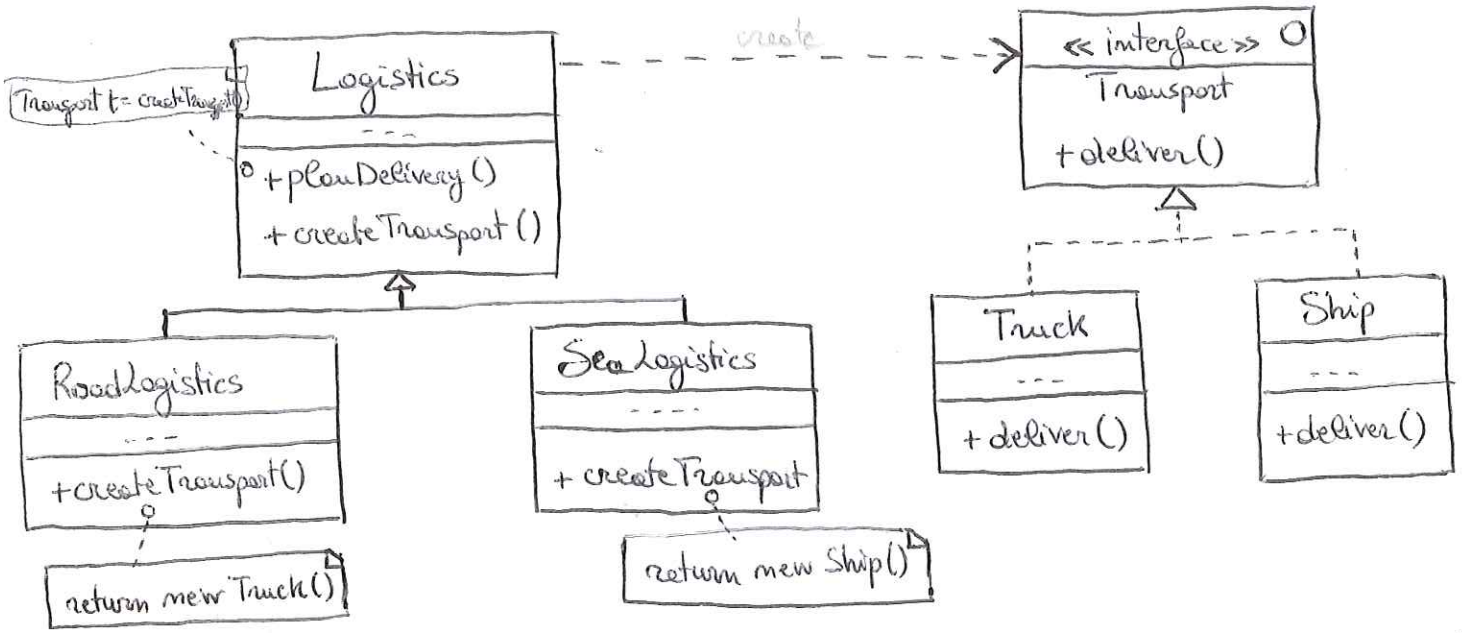
PATTERN GoF

- **FACTORY METHOD :**

Tipo: CREAZIONALE (Class)

Problema: Supponiamo di avere un sistema che gestisca trasporti tramite "Truck"; se in futuro vogliamo gestirli anche tramite "Ship", come facciamo a non dover riscrivere tutto il codice? Come creare la creazione dei mezzi di trasporto senza che siano "concretamente" legati gli uni agli altri?

Soluzione: Rimpiazzare la costruzione diretta degli oggetti con una chiamata ad un FACTORY METHOD che restituisce un generico prodotto comune (di solito interfaccia implementata da entrambi gli oggetti).



* Adesso, con tutte le SOTTOCLASSI che vogliamo, possiamo fare OVERRIDE del FACTORY METHOD (createTransport()) e cambiare la classe di prodotti create dal metodo. Tuttavia, è necessario che i prodotti abbiano un' INTERFACCIA o una CLASSE BASE comune affinché sia possibile ritornare diversi tipi di prodotti, poiché, di fatto, il tipo ritornato è quello dell' interfaccia (Transport)!

- Agli occhi del CLIENT, tutti i tipi di prodotti sono visti come Transport estratti che hanno un metodo deliver(), ma come concretamente questo metodo funziona è irrilevante.
- Il FACTORY METHOD può essere ASTRATTO, per obbligare le sottoclassi a fare Override!

• ABSTRACT FACTORY :

Tipo : CREAZIONALE (Object)

Problema : Creare FAMIGLIE di oggetti correlati senza specificare le loro classi concrete.

Per esempio, supponiamo di avere famiglie di prodotti correlati: Chair + Sofa + CoffeTable.

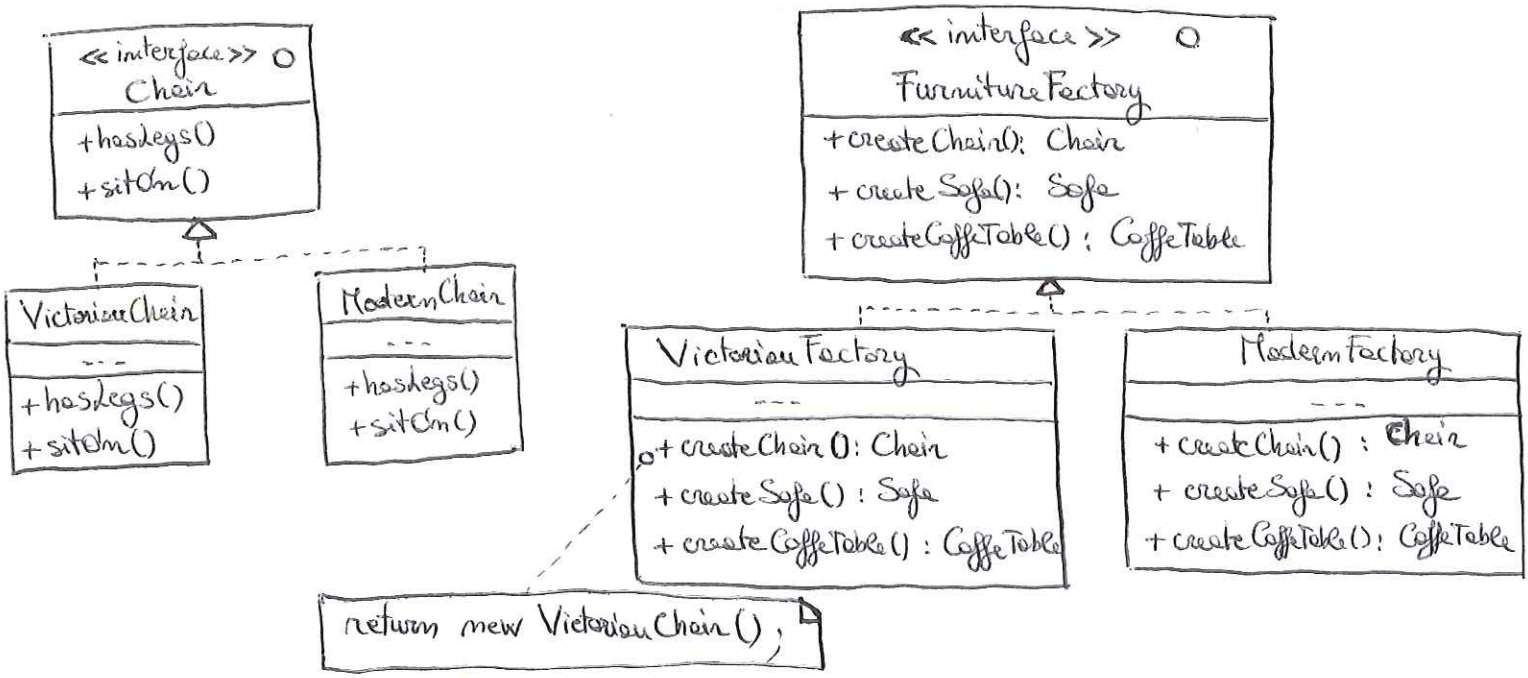
Diverse varianti di queste famiglie: Modern, Art Deco, Victorian.

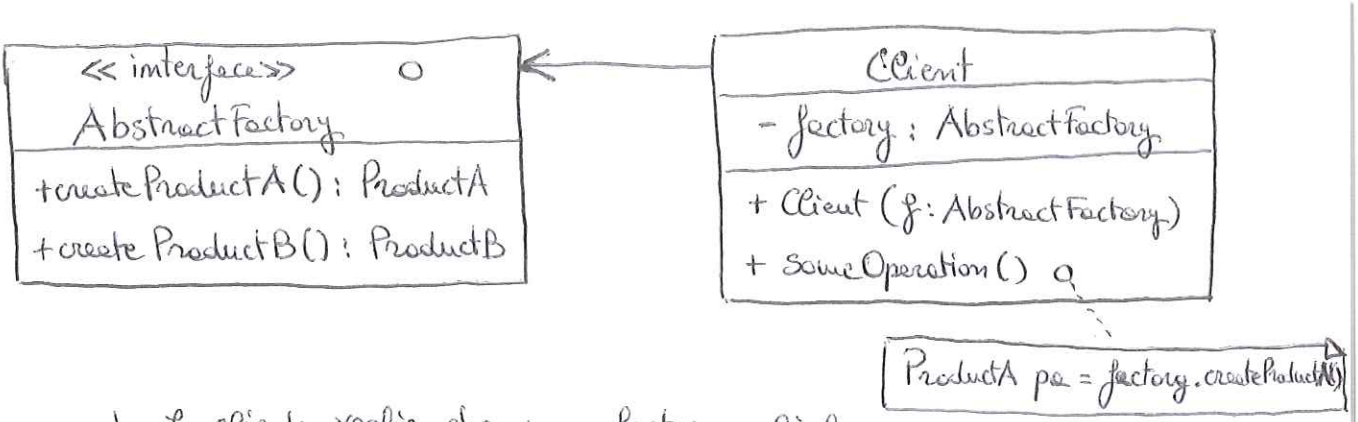
Abbiamo bisogno di un modo per creare forniture singole di oggetti di una stessa famiglia.

Inoltre, c'è spesso l'aggiunta di nuovi prodotti alle famiglie e le tipologie di famiglie cambiano spesso. Non vogliamo cambiare il codice ogni volta.

Soluzione :

1. Dichiarare INTERFACCE per ogni distinto prodotto della famiglia: Chair, Sofa, CoffeTable;
2. Creare le singole classi concrete che implementano queste interfacce: VictorianChair, ModernChair, ... ;
3. Dichiarare la ABSTRACT FACTORY : un' interfaccia con una lista di metodi di creazione per tutti i prodotti che sono parte della famiglia. Questi metodi devono ritornare un tipo di prodotto ASTRATTO, rappresentato dalle interfacce create prima (Chair, Sofa, ...).
4. Per ogni variante di famiglia, creare una CONCRETE FACTORY che implementi la abstract factory e che crei oggetti di uno SPECIFICO TIPO (ModernChair, ModernSofa, ...).





- Supponiamo che il cliente voglia che una factory gli fornisca una sedia.
  Il CLIENT NON deve avere CONOSCENZA della specifica classe FACTORY, che sia una VictorianChair o una ModernChair, il cliente la userà sempre allo stesso modo tramite l'interfaccia Chair. L'unica cosa che il cliente sa è che la sedia implementa il metodo sitOn().
  Inoltre, qualunque sia il tipo di sedia, essa farà sempre parte della stessa famiglia di Sofa e CoffeTable prodotti dalla specifica factory; infatti, il Client interagisce con la factory (attributo privato), quindi non ci sarà mai discordanza!

- Ma, se il cliente interagisce solo con le interfacce astratte, allora CHI CREA LE CONCRETE FACTORIES?

  Di solito, l'applicazione crea un oggetto CONCRETE FACTORY durante l'inizializzazione. Ma prima di ciò, l'app deve selezionare il TIPO di FACTORY sulla base della configurazione o dell'impostazioni di environment; si può ovviamente far uso del POLIMORFISMO!

PRO

- Posso essere sicuro che i prodotti presi da una factory siano compatibili
- Evito alto accoppiamento tra i concrete products e il codice del client
- SINGLE RESPONSIBILITY PRINCIPLE: mantengo un'alta coesione e buona manutenibilità.
- Posso introdurre nuove varianti di prodotti senza modificare l'esistente codice del client e degli altri prodotti

CONTRO

- Il codice può diventare più complicato per via di nuove interfacce e classi introdotte per realizzare il pattern.
- Introdurre nuovi prodotti (ad esempio, "Carpet") è difficile! Bisogna modificare la factory astratta e quelle concrete, oltre a creare nuove interfacce e prodotti concreti!

• SINGLETON :

Tipo: CREAZIONALE (Object)

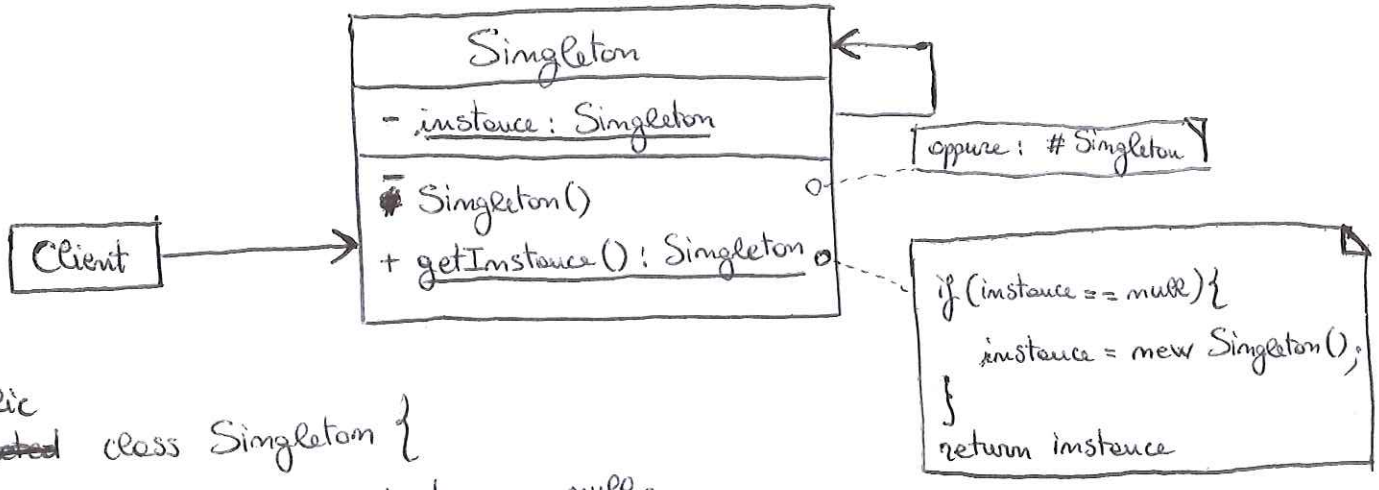
Problema: Vogliamo assicurare che una classe abbia un'UNICA ISTANZA nel sistema e vogliamo fornire un punto d'accesso globale a tale istanza.

Per esempio, il GOVERNO: un paese può avere un unico governo; inoltre, senza dare importanza ai singoli membri del governo in termini di individui, il titolo "Il Governo di X", è un punto d'accesso globale che identifica il gruppo di persone in carica.

Soluzione:

1. Rendere il costruttore privato (meglio se protected), così da impedire agli altri oggetti di usare l'operatore "new" con la classe Singleton;
2. Creare un METODO STATICO (di classe) che agisce da costruttore; questo metodo chiama il costruttore privato e salva l'istanza di Singleton in un ATTRIBUTO STATICO (di classe). Tutte le successive chiamate di questo metodo restituiscono l'unica istanza.

   C'è necessità che siano statici perché devono essere sempre chiamabili, anche se non esiste un'istanza di Singleton!




```
public
~~protected~~ class Singleton {
    private static Singleton instance = null;
    public static Singleton getInstance() {
        if (instance == null){
            this.instance = new Singleton();
        }
        return this.instance;
    }
}
```

→ * synchronized: rende l'applicazione THREAD SAFE!

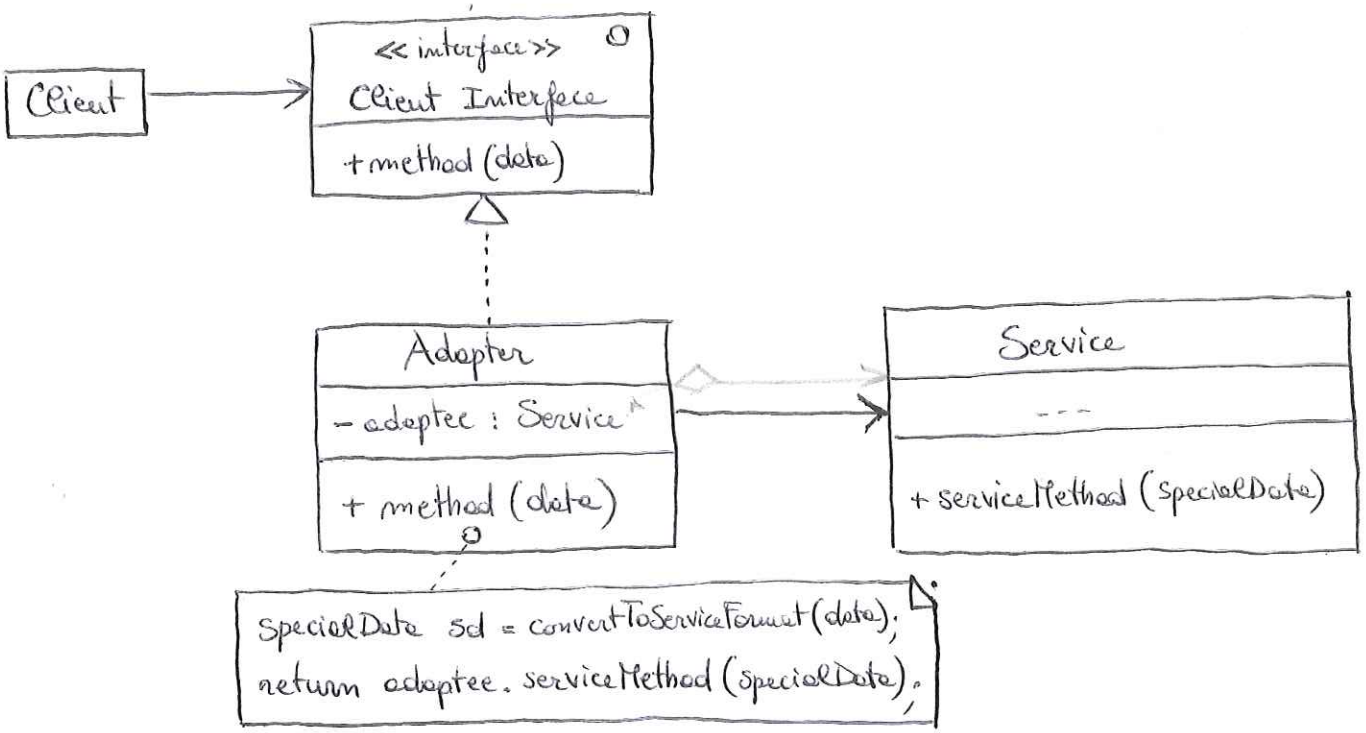
• ADAPTER :

Tipo : STRUTTURALE (Class)

Problema : Consentire a classi con INTERFACCE INCOMPATIBILI di collaborare. Per esempio, si supponga di avere un'app che lavora con dati solo in formato XML, ma si vuole aggiungere una libreria esterna che lavora con dati in formato JSON. Tale libreria è esterna e non si ha accesso al suo codice per poterla modificare : come fare?

Soluzione : Creare un ADAPTER, un oggetto speciale che converte l'interfaccia di un oggetto di modo che l'altro oggetto possa comprenderla ed usarla (XML_to_JSONAdapter).

1. L'adapter prende un'interfaccia, compatibile con uno degli oggetti esistenti;
2. Usando quest'interfaccia, l'oggetto può facilmente chiamare i metodi dell'Adapter;
3. Quando viene chiamato, l'Adapter passa la richiesta al secondo oggetto, ma nel formato che il 2° oggetto si aspetta.



- Il Client non può usare direttamente il Service, poiché ha un'interfaccia incompatibile.
- LOW COUPLING tra Client e Adapter, in quanto il Client interagisce con la sua interfaccia. Ciò permette l'aggiunta di nuovi Adapters senza modificare il codice del Client. Ciò è utile anche quando la cambiare è l'interfaccia del Service: basta cambiare solo l'Adapter.

• DECORATOR :

Tipo: STRUTTURALE (Object)

Problema: Aggiungere DINAMICAMENTE comportamenti e funzionalità ad oggetti ponendo questi oggetti in altri oggetti speciali "wrappers" che contengono i comportamenti.

Supponiamo di avere una classe Notifier che ha solo pochi campi, un costruttore e un singolo metodo send(). Il metodo può prendere come argomento un messaggio dal client e inviarlo ad una lista di emails che sono state passate al Notifier tramite il suo costruttore.
C'è un'app esterna che funge da client: crea e configura il Notifier una volta e poi lo usa ogni volta che accade qualcosa di importante.
Ad un certo punto, alcuni utenti vorrebbero ricevere messaggi anche tramite SMS, Facebook e Slack.

Ciò che viene in mente è di sfruttare l'EREDITARIETÀ e di costruire una gerarchia con classi del tipo SMSNotifier, ma ciò comporta un'esplosione del numero di classi, poiché bisogna considerare tutte le possibili combinazioni. Come fare?

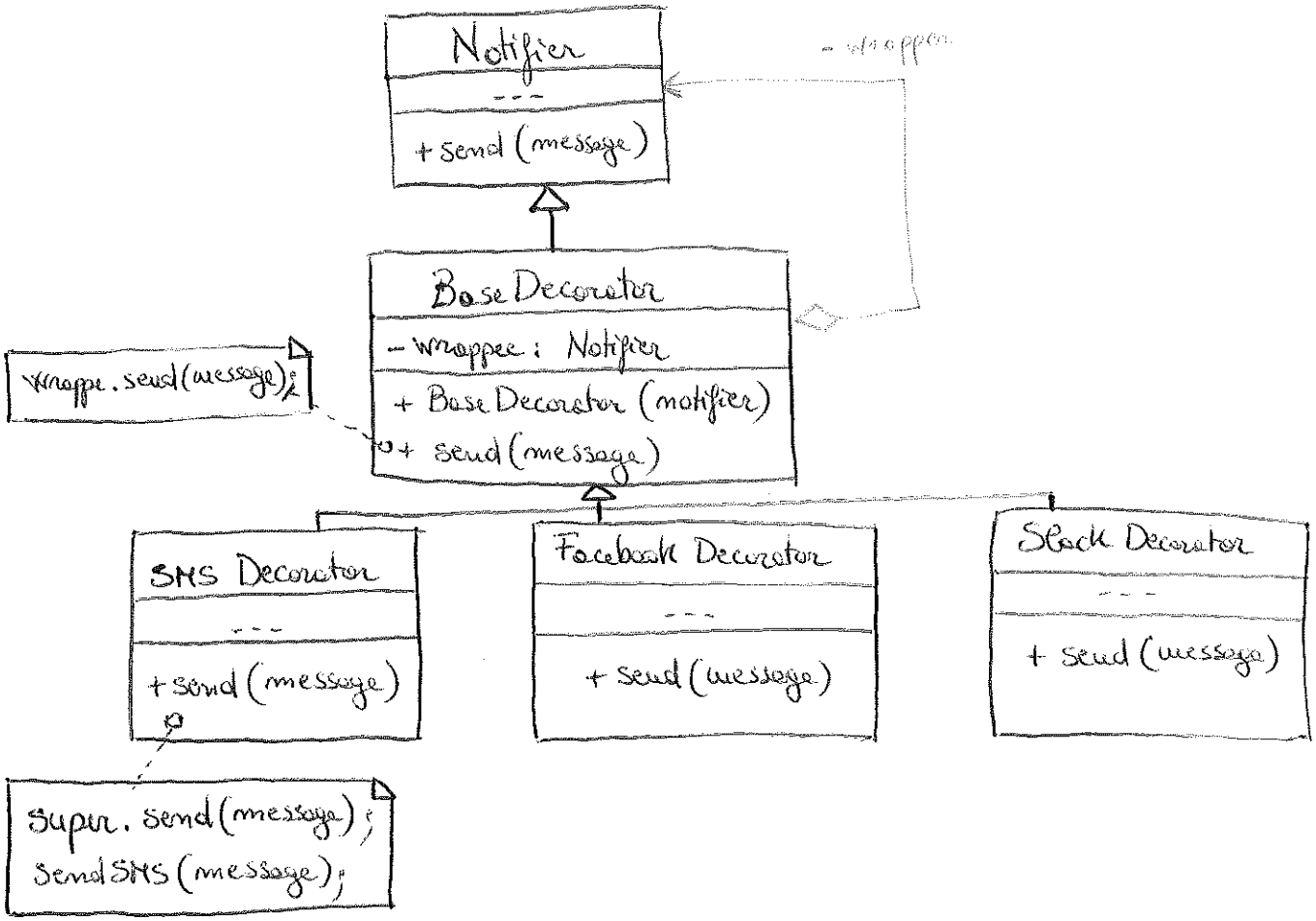
Soluzione: L'EREDITARIETÀ ha diversi limiti di cui dover essere consapevoli:

- È STATICA. Non si può cambiare il comportamento di un oggetto esistente a runtime, si può solo rimpiazzare l'intero oggetto con un altro creato da una differente sottoclasse.
- Le sottoclassi possono avere solo una classe padre; spesso l'ereditarietà multipla non è supportata.

Uno dei modi per superare questi limiti è usare l'AGGREGAZIONE o la COMPOSIZIONE invece dell'ereditarietà.
Un oggetto ha un riferimento ad un altro, cui delega del lavoro. Invece, con l'ereditarietà, l'oggetto stesso è in grado di svolgere quel lavoro, ereditando il comportamento dalle superclasse.

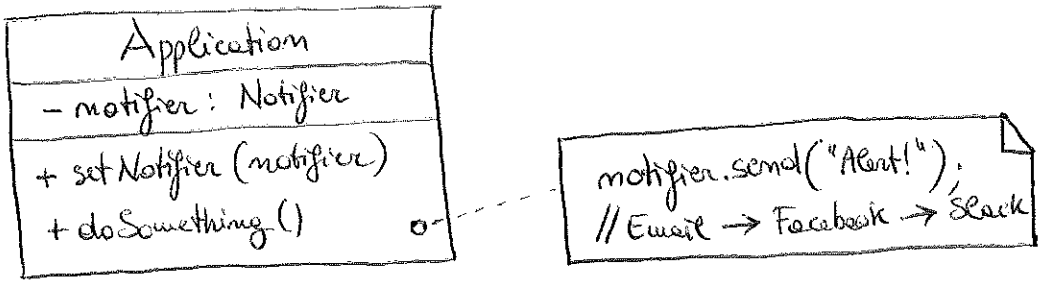
Un WRAPPER è un oggetto che può essere collegato con degli oggetti TARGET. Il wrapper contiene lo stesso insieme di operazioni del target e delega a lui tutte le richieste (sorta di meccanismo di RICORSIONE). Comunque sia, il wrapper può alterare il risultato facendo qualcosa prima o dopo aver passato le richieste al target.

Un wrapper implementa la STESSA INTERFACCIA del wrapped object e, quindi, dal punto di vista del client, i due oggetti sono identici.
Ciò permette di ricoprire un oggetto in multipli wrappers (2IVE221), aggiungendo ad esso i comportamenti combinati dei diversi possibili wrappers!

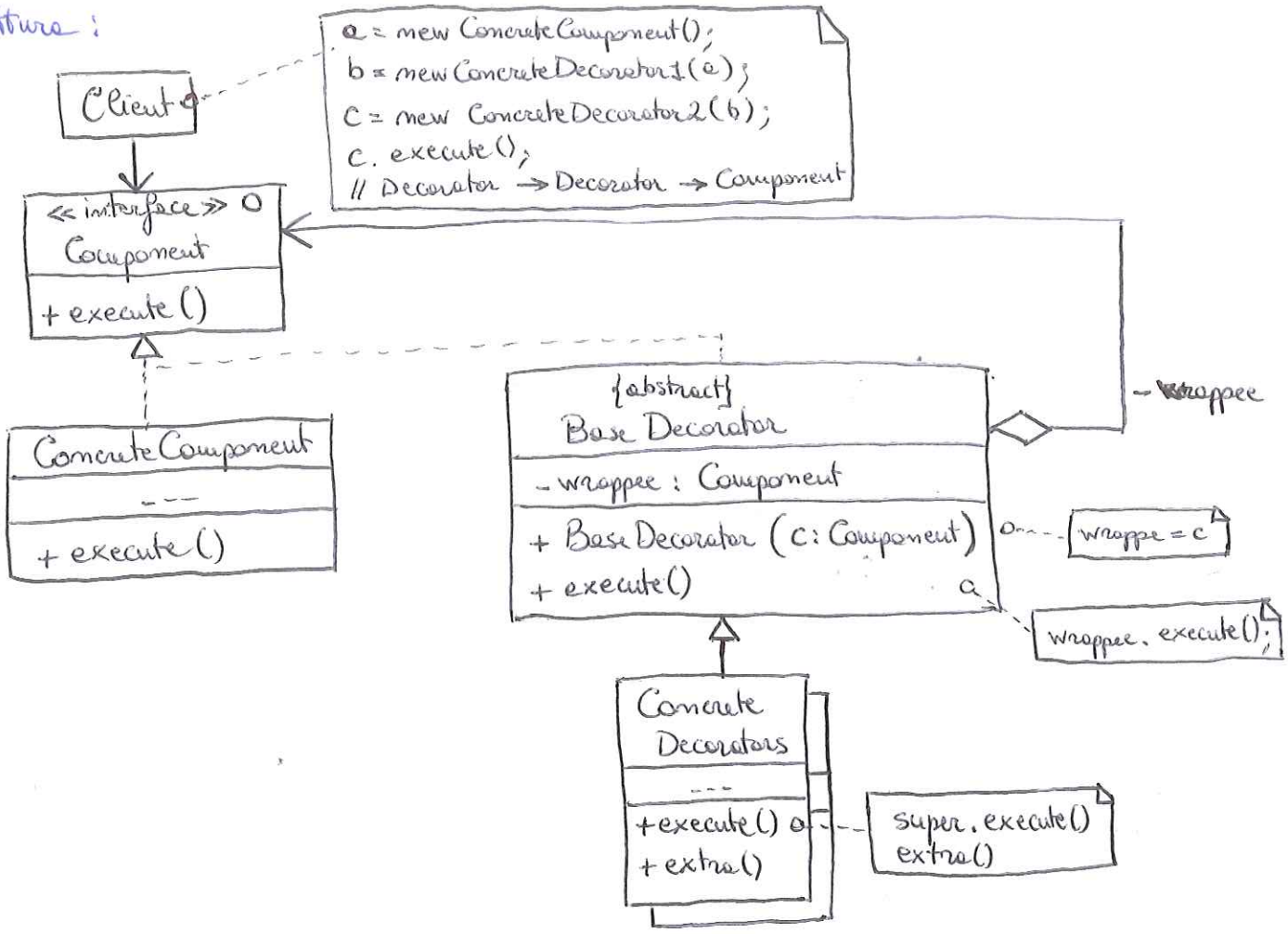


- Il Client code necessita di avvolgere un oggetto Notifier basico in un insieme di decorators in accordo con le proprie preferenze. L'oggetto risultante sarà strutturato come uno STACK:

```
stack = new Notifier();
if (facebookEnabled)
    stack = new FacebookDecorator(stack);
if (slackEnabled)
    stack = new SlackDecorator(stack);
app.setNotifier(stack);
```



Struttura:



- ConcreteComponent: è la classe di oggetti WRAPPED, definisce il comportamento di base, alterabile tramite i decorators.

- BaseDecorator: ha un attributo wrappee per riferire l'oggetto WRAPPED, da dichiarare con il tipo dell'INTERFACCIA, così che esso possa essere sia un ConcreteComponent che un Decorator!
  Il BaseDecorator delega tutte le operazioni all'oggetto wrapped.

- ConcreteDecoratorX: definisce comportamenti extra che possono essere aggiunti DINAMICAMENTE ai componenti.
  Fa OVERRIDE dei metodi del BaseDecorator ed esegue il loro comportamento prima o dopo di chiamare i metodi del padre tramite super() → AVVIA LA "RICORSIONE"

- Client: può fare wrap ("avvolgere") di componenti in livelli multipli di decorators, in quanto esso interagisce con tutti gli oggetti tramite l'interfaccia!

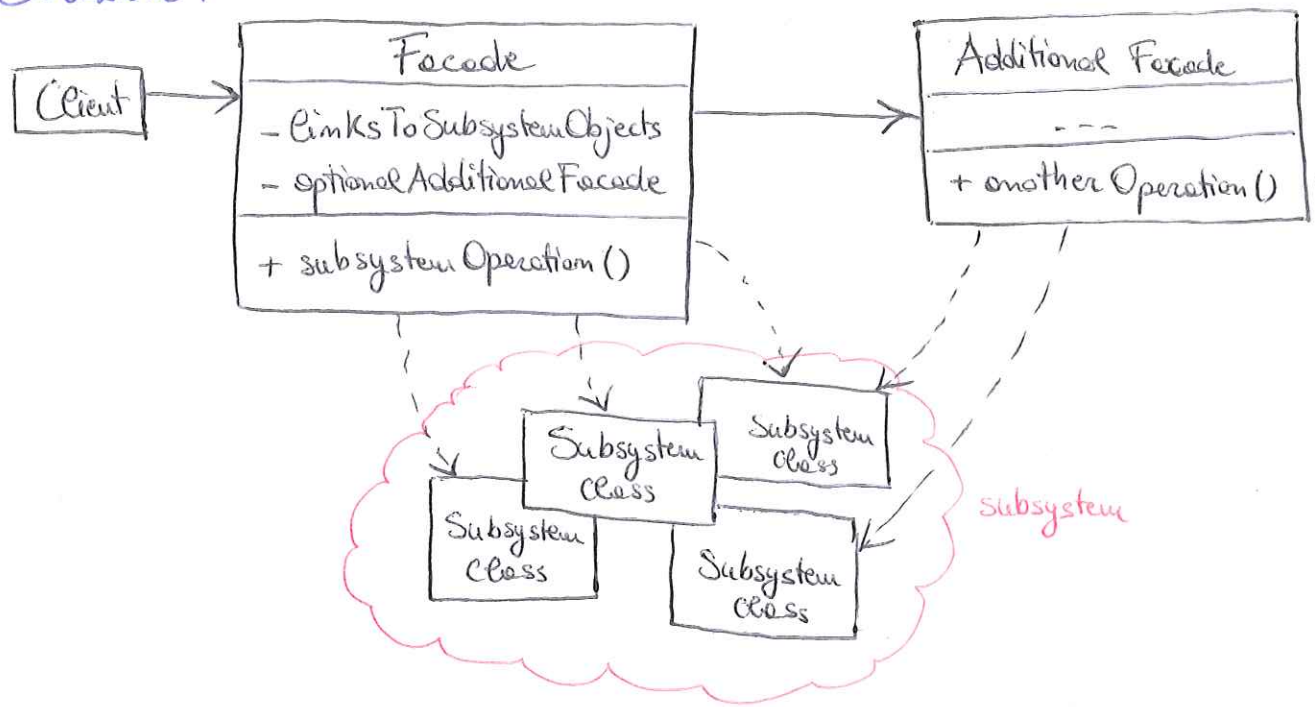
* Il BaseDecorator è dichiarato ABSTRACT per evitare che venga istanziato, in quanto si vogliono solo concrete decorators!

- **FAÇADE :**

Tipo: STRUTTURALE (Object)

Problema: Fornire un' INTERFACCIA UNICA ad una libreria, un framework o ad un insieme complesso di classi e/o interfacce di un sottosistema.
Per esempio, una FAÇADE puo' essere la compilazione in C, poiché è un' unica interfaccia che unifica ed esegue in ordine:
compilazione sorgente + compilazione librerie + linking + generazione exe.

Soluzione:



- Client: usa il Facade invece di invocare direttamente le sottoclassi del sottosistema (LOW COUPLING & HIGH COHESION);

- Facade: fornisce un accesso facilitato ad una specifica parte della funzionalità del sottosistema, esso sa dove dirigere le richieste del client e come coordinarle;

- Additional Facade: puo' essere creato per prevenire l' "inquinamento" di un singolo facade con funzionalità scorrelate che potrebbero farlo diventare un' altra struttura complessa.
Additional Facades possono essere usati sia dal Client che da altri Facades!

- OBSERVER :

Tipo: COMPORTAMENTALE (Object)

Problema: Definire un meccanismo di SOTTOSCRIZIONE per notificare una moltitudine di oggetti circa alcuni eventi che accadono all'oggetto che loro stanno osservando. Tuttavia, si vuole mantenere un ALTO livello di DISACCOPPIAMENTO tra osservatori e osservato!

Supponiamo di avere 2 tipi di oggetti: un Customer e uno Store. Il Customer è interessato ad un particolare tipo di prodotto, presto disponibile nello Store. Il Customer potrebbe visitare tutti i giorni lo Store per controllare, ma farebbe molti viaggi a vuoto.
Alternativamente, lo Store potrebbe inviare tonnellate di emails a tutti i Customers ogni volta che è disponibile un nuovo prodotto. Tuttavia, ciò comporta un grande spreco di risorse per lo Store e notifica anche molti Customers non interessati, il che può farli irritare. Come fare?

Soluzione: L'oggetto cui gli altri oggetti sono interessati è spesso chiamato "Subject", ma poiché notifica gli altri è spesso detto anche "Publisher". Tutti gli oggetti interessati alle notifiche di cambiamento del Publisher sono detti "Subscribers" oppure "OBSERVERS".

Dunque, il meccanismo di sottoscrizione richiede che il Publisher abbia una lista di Observers e diversi metodi che permettono di notificare, aggiungere o rimuovere Observers alla lista.

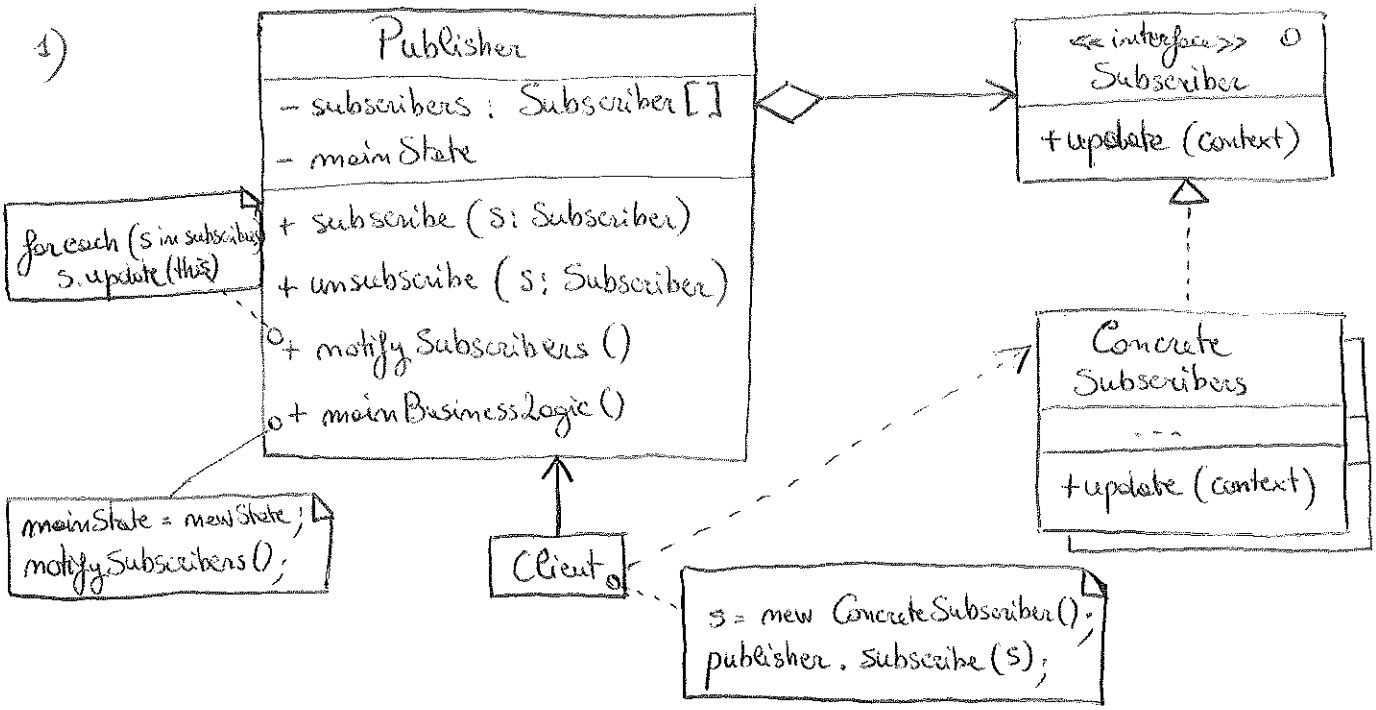
Quando succede qualcosa al Publisher, esso scorre tutte le sue liste di Observers (Subscribers) e chiama lo specifico metodo di notifica degli observers (update()) !

Se ci sono diversi tipi di subscribers, è CRUCIALE che tutti i subscribers implementino le stesse interfacce, così che il Publisher possa interagire solo con l'interfaccia e mantenere un BASSO ACCOPPIAMENTO → LOW COUPLING

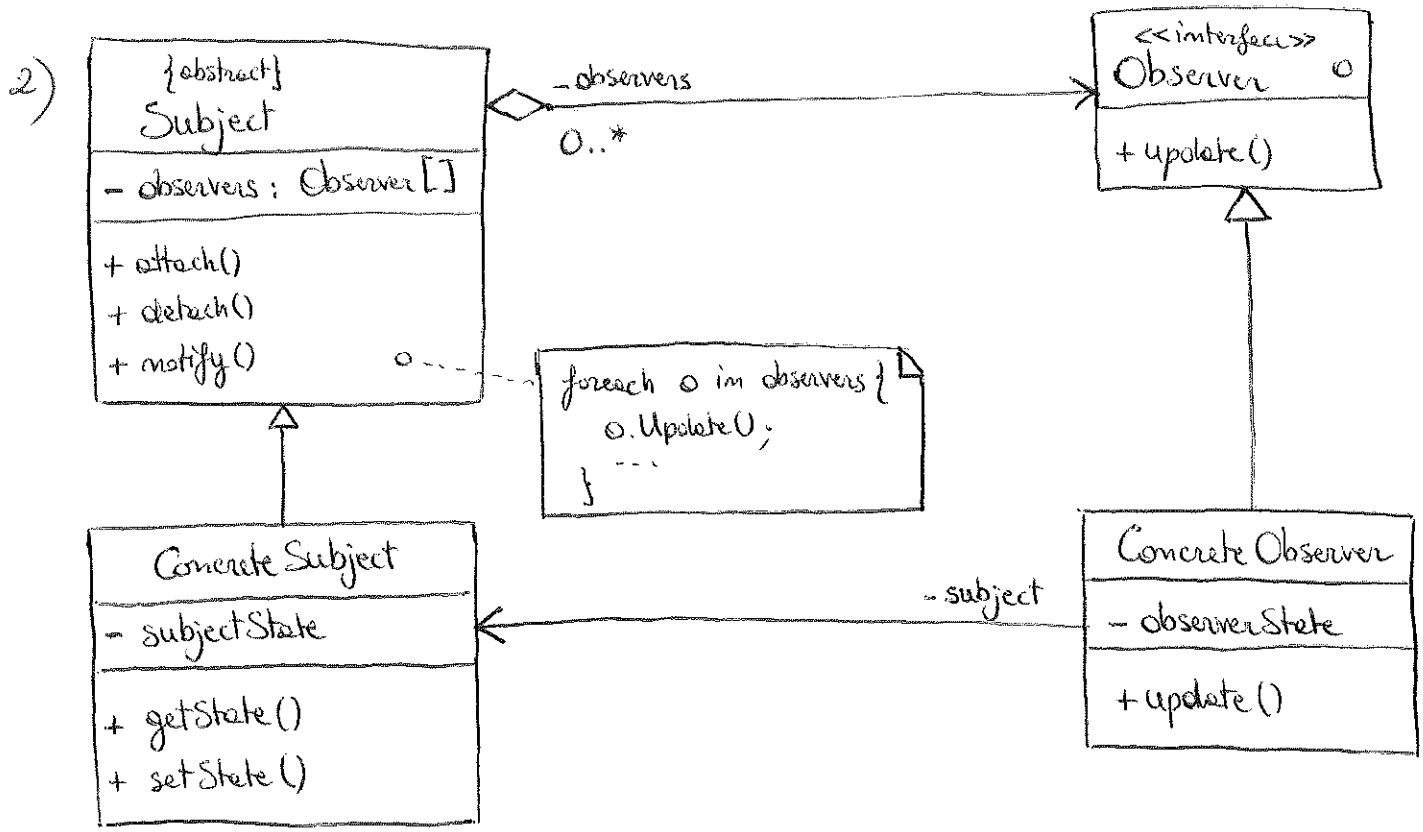
Inoltre, se ci sono più Publishers e i subscribers possono essere interessati a più tipi di Publisher, allora anche qui è bene creare un'interfaccia BasePublisher implementata dai vari ConcretePublishers. Tuttavia, dovendo registrare i subscribers, deve essere una ABSTRACT CLASS.

Strutture:

1)



2)



- C'è la relazione tra ConcreteObserver e ConcreteSubject perché potrebbero esserci diversi ConcreteSubjects e volerci diversi ConcreteObservers con metodi specifici per ottenere lo stato dello specifico Subject.
- Nel Subject, il metodo notify() è PUBLIC! → Ciò comporta che può essere notificato un cambiamento da chiunque, anche se il cambiamento non è realmente avvenuto (SIDE EFFECT).

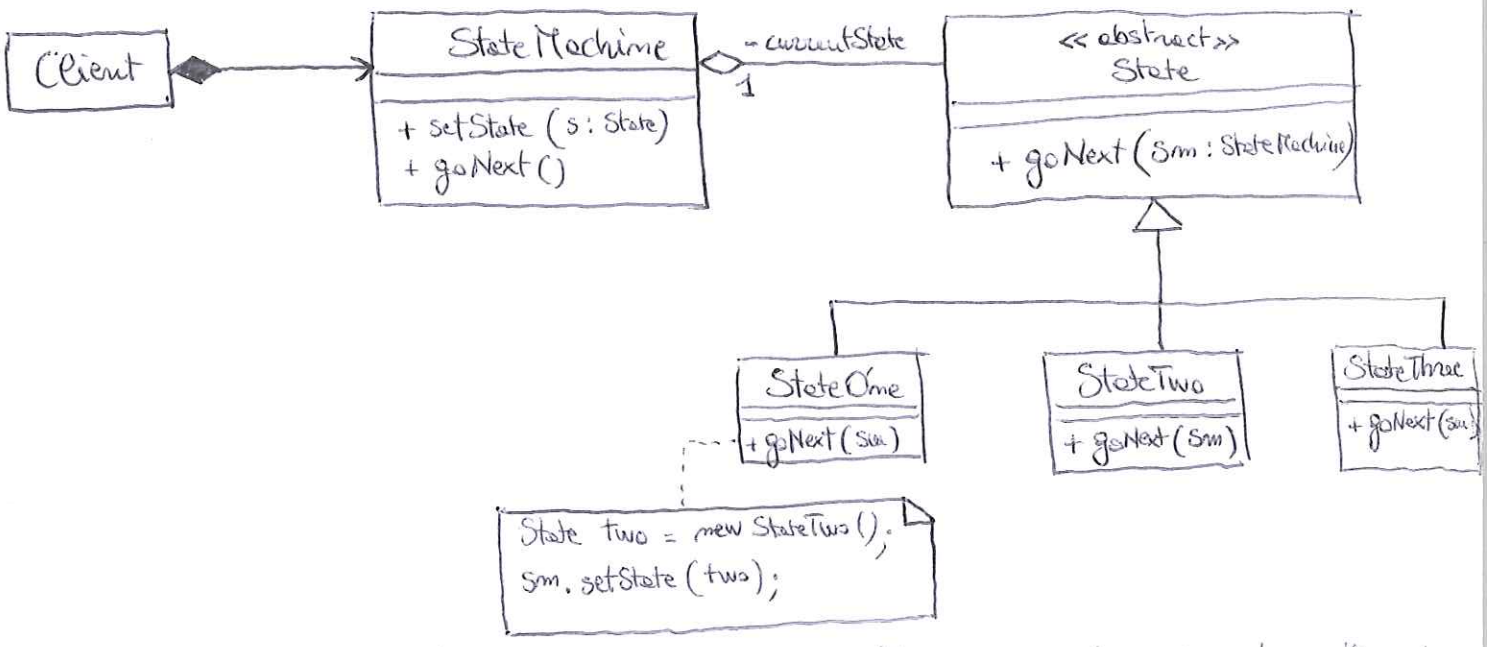
  Conviene mettere la visibilità del metodo notify() a PROTECTED!

• STATE :

Tipo: COMPORTAMENTALE (Object)

Problema: Cambiare il comportamento di oggetti a run-time, a seconda dello stato in cui si trova l'oggetto. Gli stati possibili sono in numero finito e noti a priori.
La soluzione di statement condizionali (if-else / switch) NON va bene se ci sono molteplici stati e combinazioni, in quanto rende il codice molto poco manutenibile e i cambiamenti hanno un grosso impatto!

Soluzione: Definire una classe per ogni possibile stato, di modo che implementino tutte le stesse interfacce che espone operazioni comuni, ma il cui specifico comportamento è implementato nello specifico stato. Definire una classe StateMachine ("context"), che rappresenta il contratto d'uso con il client, mantiene lo stato corrente e delega le operazioni invocate dal client allo specifico stato.



* StateMachine NON può essere un Singleton! Il comportamento deve dipendere dalla singola istanza di SM e dal suo stato. Ogni client ha un suo flusso logico indipendente dagli altri, quindi deve avere una propria SM! (COMPOSIZIONE e non aggregazione). Bisogna passare la SM come PARAMETRO!

* ALTA MANUTENIBILITÀ → aggiungere o togliere stati o cambiare transizioni ha un piccolo impatto e non si ripercuote sull'interfaccia esposta al client

* ALTO COUPLING tra le classi che rappresentano i veri stati concreti, c'è bisogno che ognuno di queste classi abbia una buona conoscenza delle altre.