

2 tipi

venerdì 30 giugno 2023

21:43

praticamente sono di due tipi: - responsabilità di conoscere, es conoscere i suoi dati, oppure sapere come fare certe operazioni e responsabilità di fare (es. creare un oggetto oppure calcolare un valore)

21:39

e tramite i suoi metodi le fa

21:39

NOCCIOLI GENE PALLI PATTERN GRASP

Il nostro obiettivo è quello di rendere la progettazione modulare. Quindi il software deve essere decomposto in un insieme di elementi software (moduli) coesi e debolmente accoppiati

PATTERN CREATOR

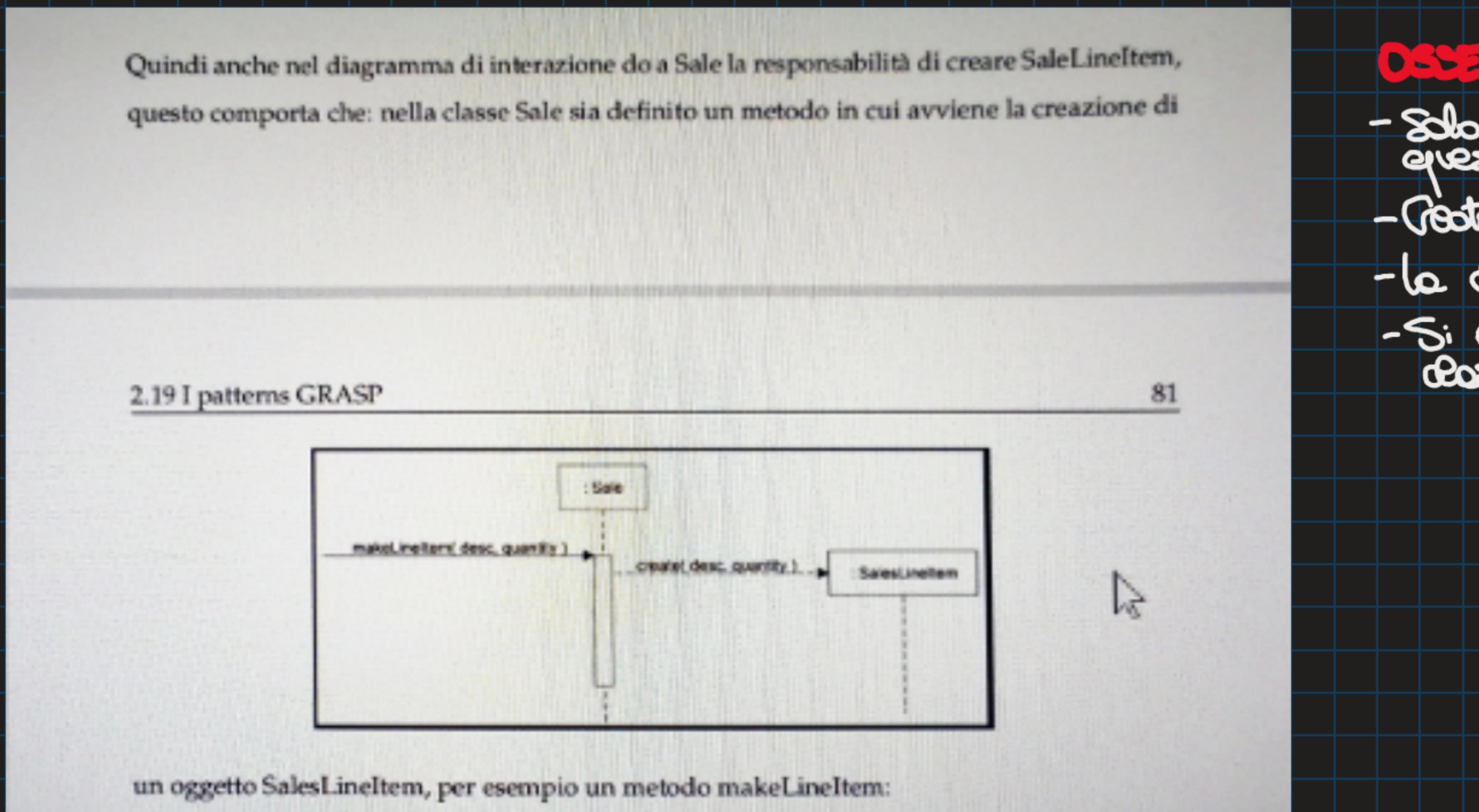
Pattern Creator

Nome: Creator (Creatore)

Problema: Chi crea un oggetto A? Ovvvero, chi deve essere responsabile della creazione di una nuova istanza di una classe?

Soluzione: Assegna alla classe B la responsabilità di creare un'istanza della classe A se una delle seguenti condizioni è vera (più sono vere meglio è):

- B "contiene" o aggrega con una composizione oggetti di tipo A
- B registra A²
- B utilizza strettamente A
- B possiede i dati per l'inizializzazione di A, che saranno passati ad A al momento della sua creazione (pertanto B è un Expert rispetto alla creazione di A)



un oggetto SalesLineItem, per esempio un metodo makeLineItem:

PROBLEMA

chi è responsabile della creazione di una nuova istanza di una classe

SOLUZIONE

ha la responsabilità chi rispetta almeno una di queste condizioni:
Supponiamo B ha responsabilità della classe A

- B è l'unico che utilizza la classe A
- B possiede i dati per l'inizializzazione di A → B EXPERT rispetto alla creazione di A
- B ha una variabile di tipo A a cui è assegnato un oggetto di tipo A
- B è una composizione di istanze A

COMPOSIZIONE: tipo forte di aggregazione PART-OF

**Osservazioni**

- Solo chi crea effettivamente altre classi collegandole con l'istanza da creare questo permette di diminuire le dipendenze tra classi
- Creator corrisponde a Low Coupling, fornisce un accoppiamento basso, minori dipendenze di manutenzione e maggiori opportunità di uso.
- La classe creatrice deve essere visibile al creatore
- Si devono usare classi di supporto se la creazione è a richiesta o se una proprietà esterna condiziona la scelta della classe creatrice

Information expert

venerdì 30 giugno 2023 20:11

Pattern Expert

Nome: Information Expert (Esperto delle Informazioni)

Problema: Qual è un principio di base, generale, per l'assegnazione di responsabilità agli oggetti?

Soluzione: Assegna una responsabilità alla classe che possiede le informazioni necessarie per soddisfarla, all'esperto delle informazioni, ovvero alla classe che possiede le informazioni necessarie per soddisfare la responsabilità

PROBLEMA

su quale basi diamo la responsabilità a degli oggetti?

SOLUZIONE

Assegniamo una responsabilità all'oggetto che possiede le informazioni necessarie per soddisfare, al INFORMATION EXPERT

PROBLEMI PATTERN

Chi solva un oggetto INFORMATION EXPERT?

dovrebbe autoaddiversi contenendo le

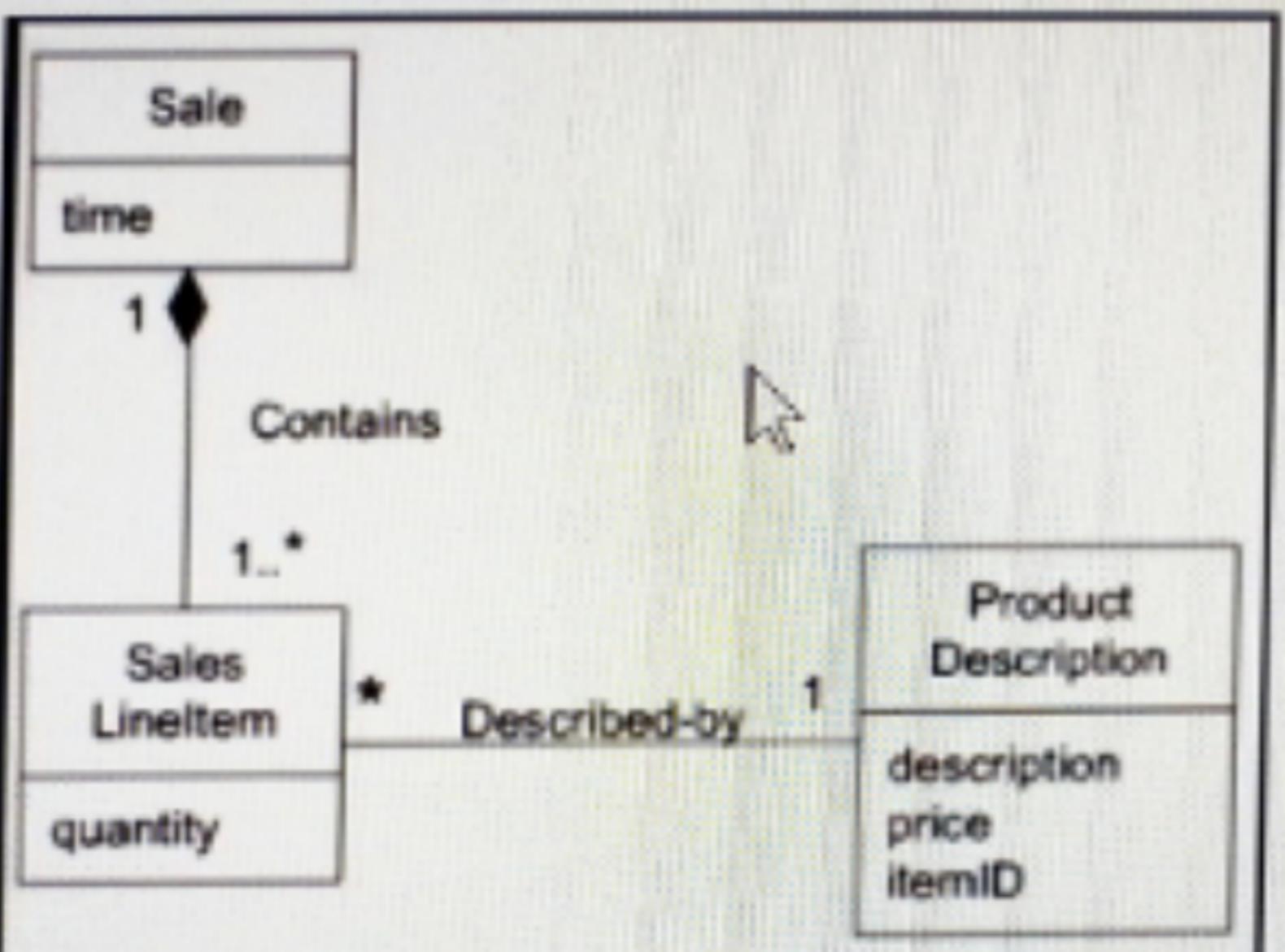
informazioni su se stesso implicherebbe:

1. GESTIONE
2. ACCOPPIAMENTO
3. DOPPLICAZIONE

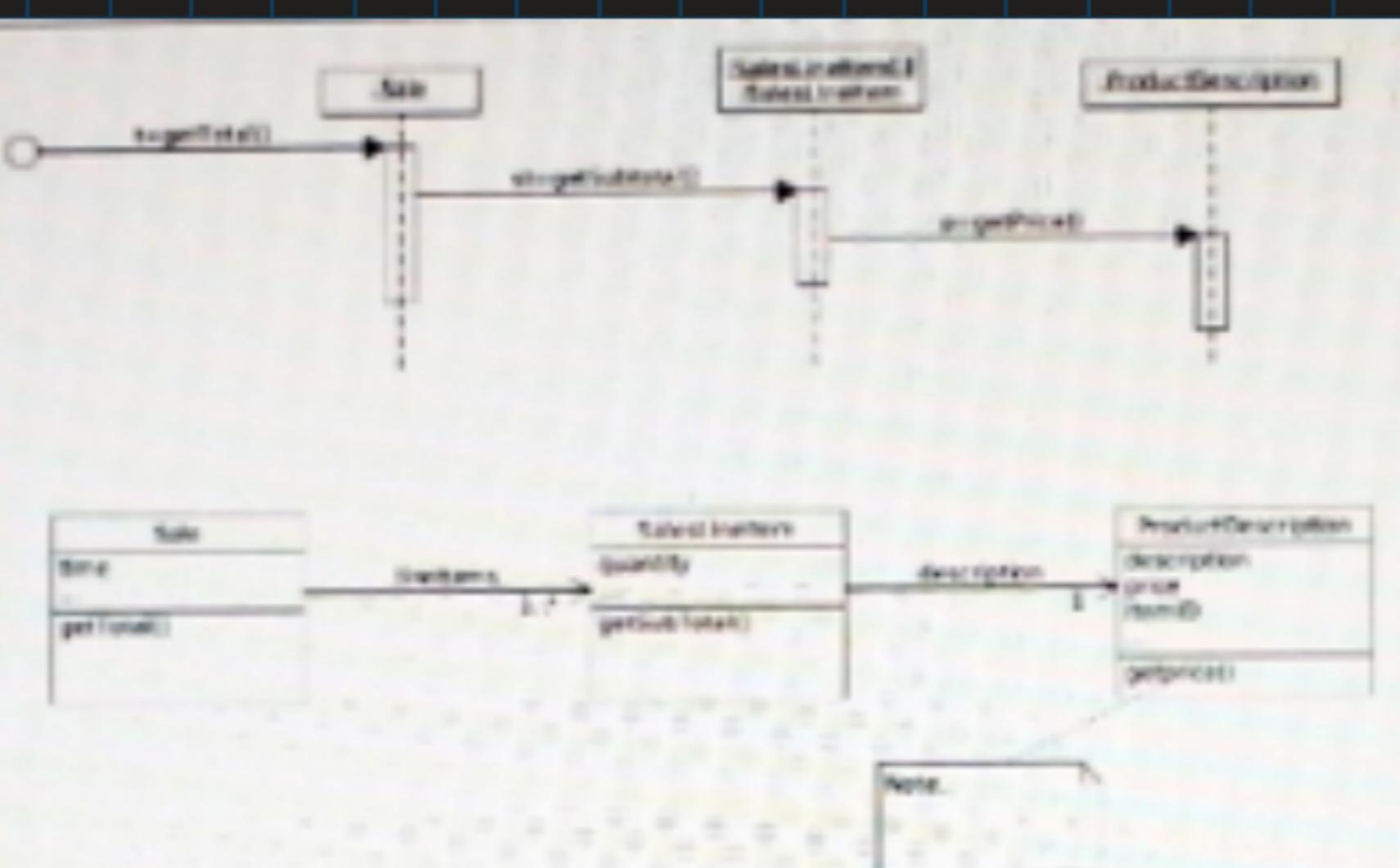
La soluzione per capire se è quindi corretto applicare questo pattern è data dae pattern Low Coupling

Una responsabilità necessita di informazioni per essere soddisfatta: informazioni su altri oggetti, sullo stato di un oggetto, sul mondo che circonda l'oggetto, **informazioni che l'oggetto può ricavare.**

La soluzione in base al pattern Information Expert è Sale, perché occorre conoscere tutte le istanze SalesLineItem della vendita (per sapere il totale devo conoscere tutte le voci della vendita) e la somma dei relativi totali parziali. Un'istanza Sale li contiene, è un esperto delle informazioni per questo compito.



Si assegna la responsabilità a Sale di conoscere il suo totale, esprimendo questa responsabilità con un metodo chiamato getTotal():



Chi maggio chi chi ti contiene
conosce le informazioni.
Fanno parte di lui

Low Coupling

venerdì 30 giugno 2023 20:38

Pattern Low Coupling

Nome: **Low Coupling** (Accoppiamento Basso)

Problema: Come ridurre l'impatto dei cambiamenti? Come sostenere una dipendenza bassa, un impatto dei cambiamenti basso e una maggiore opportunità di riuso?

Soluzione: Assegna le responsabilità in modo tale che l'accoppiamento (non necessario) rimanga basso. Usa questo principio per valutare le alternative.

L'accoppiamento è una misura di quanto
semplicemente è coinvolto un elemento con
altri elementi.

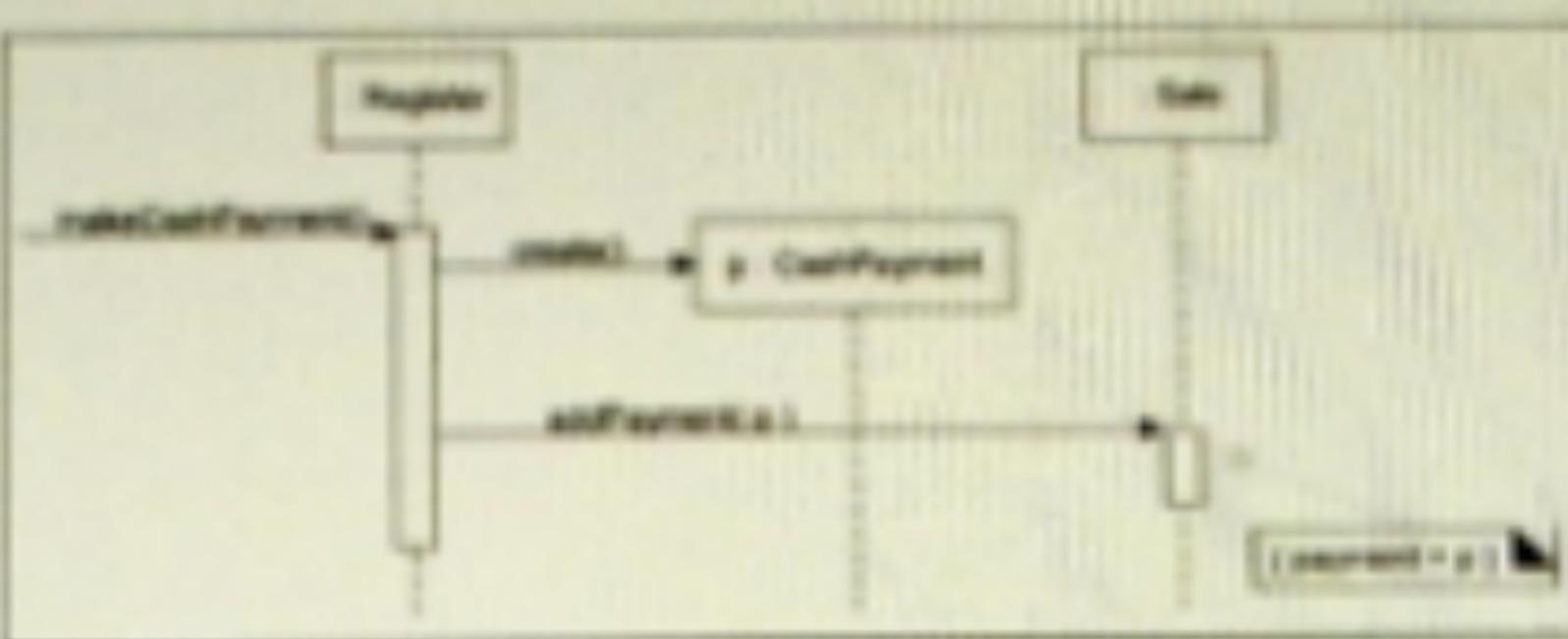
Sceglio la soluzione con un basso accoppiamento

Una classe con un accoppiamento alto (o forte) dipende da molte altre classi. Quando una classe ha un alto accoppiamento possono esserci problemi com:

- I cambiamenti nelle classi correlate, da cui ~~queste~~ dipendono, obbligano a cambiamenti locali anche ~~in queste classi~~
che possiedono molte classi che dipendono
- Le classi che dipendono da altre sono più difficili da comprendere in isolamento
- il loro uso richiede la presenza aggiuntiva delle classi da cui dipendono

Esempio di applicazione pattern Low Coupling

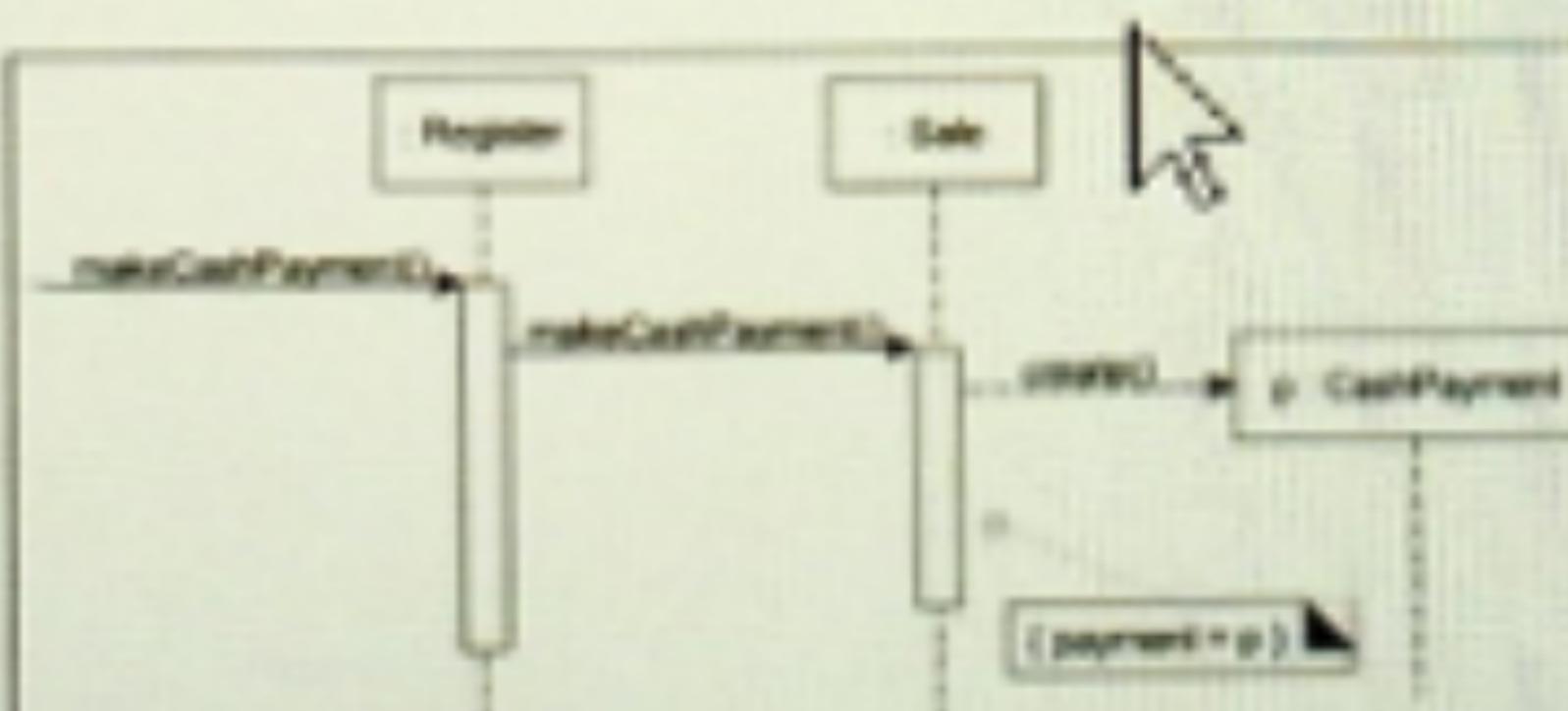
Supponiamo di essere in questa situazione in cui con il pattern **Creatore** si sceglie Register come **creatore** di Payment (per la responsabilità di registrare i pagamenti), uso addPayment(p) per comunicare con Sale (perché nel modello di dominio la SALE è associata al pagamento):



Per determinare gli accoppiamenti si contano le dipendenze, che sono direzionali:

- Register -> CashPayment, questa è data dalla create(), *che crea CashPayment quindi vi è una dipendenza*.
- Register -> Sale, perché register invoca il metodo addPayment di Sale.
- Sale -> CashPayment, perché Sale memorizza p, un'istanza di cashPayment.

Attenzione però notiamo che Sale deve comunque conoscere Payment, per cui l'accoppiamento tra Payment e Register è inutile, di conseguenza modificando il progetto possiamo togliere la dipendenza, Register -> CashPayment, andando a far creare CashPayment alla Sale, perché tanto anche lei dipende da CashPayment, tanto vale farla creare a Lei:



Questo progetto va preferito rispetto al precedente perché mantiene un accoppiamento complessivo più basso.

Low Coupling è un principio di valutazione da utilizzare in parallelo ad altri pattern.

PROBLEMA

Come ridurre l'impatto dei cambiamenti? Basso IMPATTO ALTO RIUSO

SOLUZIONE

Assegna le responsabilità in modo tale che l'accoppiamento rimanga basso.

FORME COMUNI DI ACCOPPIAMENTO

- la classe X ha un attributo (una variabile d'istanza o un dato membro) di tipo Y o referenzia un'istanza di tipo Y o una collezione di oggetti Y.
- un oggetto di tipo X richama operazioni o servizi di un oggetto di tipo Y.
- il tipo X ha un metodo che contiene un elemento (parametro, variabile locale oppure tipo di ritorno) di tipo Y o che referenzia un'istanza di tipo Y.
- è un'interfaccia, e la classe X implementa questa interfaccia.

L'accoppiamento eccessivo è l'impatto dei cambiamenti,
se è alto ma con elementi stabili o con elementi
peruviani va bene comunque
Immagini secondo il principio di Low coupling il problema
non è di per sé l'accoppiamento alto, ma l'accoppiamento
alto con elementi instabili

High Cohesion

venerdì 30 giugno 2023 21:18

Pattern High Cohesion

Nome: High Cohesion (Cohesione Alta)

Problema: Come mantenere gli oggetti focalizzati, comprensibili e gestibili e, come effetto collaterale, sostenere Low Coupling?

Soluzione: Assegna le responsabilità in modo tale che la coesione rimanga alta. Usa questo principio per valutare alternative.

La coesione è una misura di quanto i statementi
sono focalizzati e concentrati nel rappresentato di un
elemento sostitutivo

↳ Sono obiettivo diverse funzioni, dentro uno
uovo devono esserci solo funzioni coerenti
con un dato obiettivo

Livelli di coesione

- Coesione molto bassa: una classe è la sola responsabile di molte cose in aree funzionali molto diverse. (Da evitare).
- Coesione bassa: una classe ha da sola la responsabilità di un compito complesso in una sola area funzionale. (Da evitare).
- Coesione alta: una classe ha responsabilità moderate in un'area funzionale e collabora con altre classi per svolgere i suoi compiti
- Coesione moderata: una classe ha, da sola, responsabilità in poche aree diverse, che sono logicamente correlate al concetto rappresentato dalla classe, ma non l'una all'altra. Esempio: una classe Company che è completamente responsabile di conoscere i suoi dipendenti e conoscere le proprie informazioni finanziarie, queste non sono aree correlate anche se entrambe sono logicamente legate al concetto di una Company.

Una classe con coesione alta ha un numero di metodi relativamente basso, con delle funzionalità altamente correlate e focalizzate, e non fa troppo lavoro.

PROBLEMA

Come manteniamo oggetti focalizzati, comprensibili e gestibili e, quindi, sostenere anche un low Coupling?

SOLUZIONE

Assegnare le responsabilità in modo tale che la coesione rimanga alta

Esempio di applicazione del pattern High Cohesion

Prendi l'immagine di prima (quella prima di applicare low coupling per valutare il secondo progetto come migliore) dove abbiamo applicato il pattern **Creator** a Register, poi abbiamo add Payment per far sì che Register comunichi con Sale, se si continua a rendere la classe Register responsabile di eseguire una parte del lavoro e l'intero lavoro relativo a sempre più operazione di sistema, essa diventerà sempre più carica di compiti, e diventerà non coesa. Applicando High Cohesion, confrontando e valutando i due progetti, si preferisce il secondo progetto dove la responsabilità della creazione del pagamento è delegata alla Sale e sostiene una coesione più alta di Register. Questo progetto che sostiene una coesione alta e un accoppiamento basso, è da preferire.

Nota che sia High Cohesion che Low Coupling sono principi di VALUTAZIONE da usare in parallelo ad altri pattern.

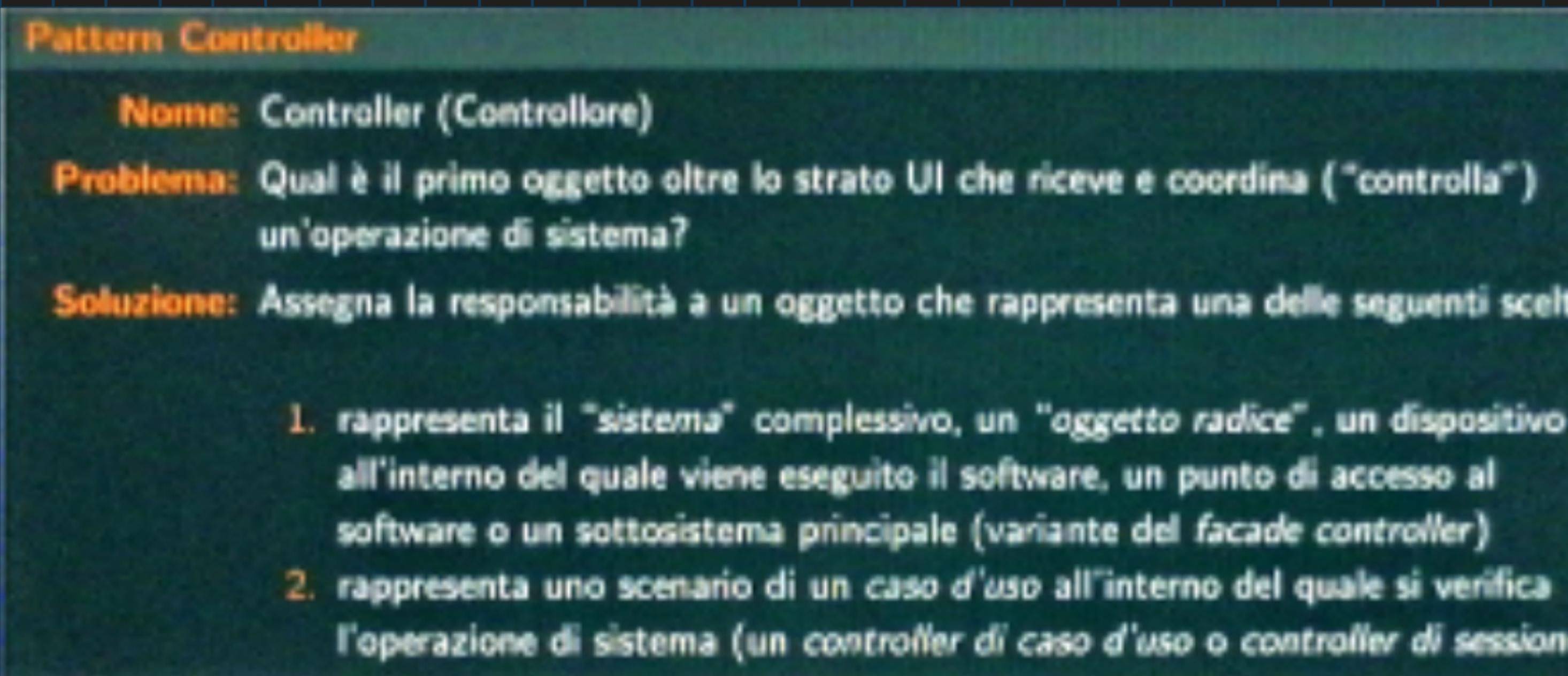
Controller

venerdì 30 giugno 2023 21:54

Dobbiamo collegare i vari strati della nostra applicazione, come controller e interfaccia con lo strato di dominio che vogliamo tenere separato.

L'interfaccia grafica intercette l'evento dell'utente, i.e. poter comandare decide chi è l'oggetto che deve ricevere e coordinare l'operazione intercettata.

↳ L'istante



Se controller è un polmone di delega e da una soluzione come connettere lo strato UI allo strato del dominio applicativo

Il controller deve coordinare le operazioni di sistema, controlla le attività, mantenere una sessione, mantenere gli oggetti correnti della sessione. Ad esempio la vendita corrente dove la metto? All'interno del controller.

Un controller soffre di una coesione bassa, violando il principio High Cohesion.

Utilizzare la stessa classe controller per tutti gli eventi di sistema di un unico caso d'uso, in questo modo il controller può conservare le informazioni sullo stato del caso d'uso.

Inoltre un controller è molto utile per identificare gli eventi di sistema fuori sequenza rispetto all'SSD, banalmente utilizzando ad esempio una variabile booleana "done" che è a true se un'operazione è stata eseguita (ad esempio quando viene fatta la endSale, imposto a true la variabile done).

Se controller MVC delega le richieste di lavoro dell'utente al controller grafico del dominio

```
public void actionPerformed(ActionEvent e) {
    // Transformer è una classe di utilità per
    // trasformare le stringhe in altri tipi di dati
    // perché gli elementi JTextField della GUI
    // gestiscono solo stringhe
    ItemID id = Transformer.toItemID(getTXT_ID().getText())
    int qty = Transformer.toInt(getTXT_QTY().getText());

    // qui si supera il confine tra lo strato UI
    // e lo strato del dominio delegando al 'controller'
    // >> QUESTA È L'ISTRUZIONE CHIAVE <<
    register.enterItem(id, qty);
}
} // fine della chiamata a addActionListener
```

PROBLEMA

Quel è il primo oggetto oltre lo strato UI che riceve e coordina un'operazione di sistema?

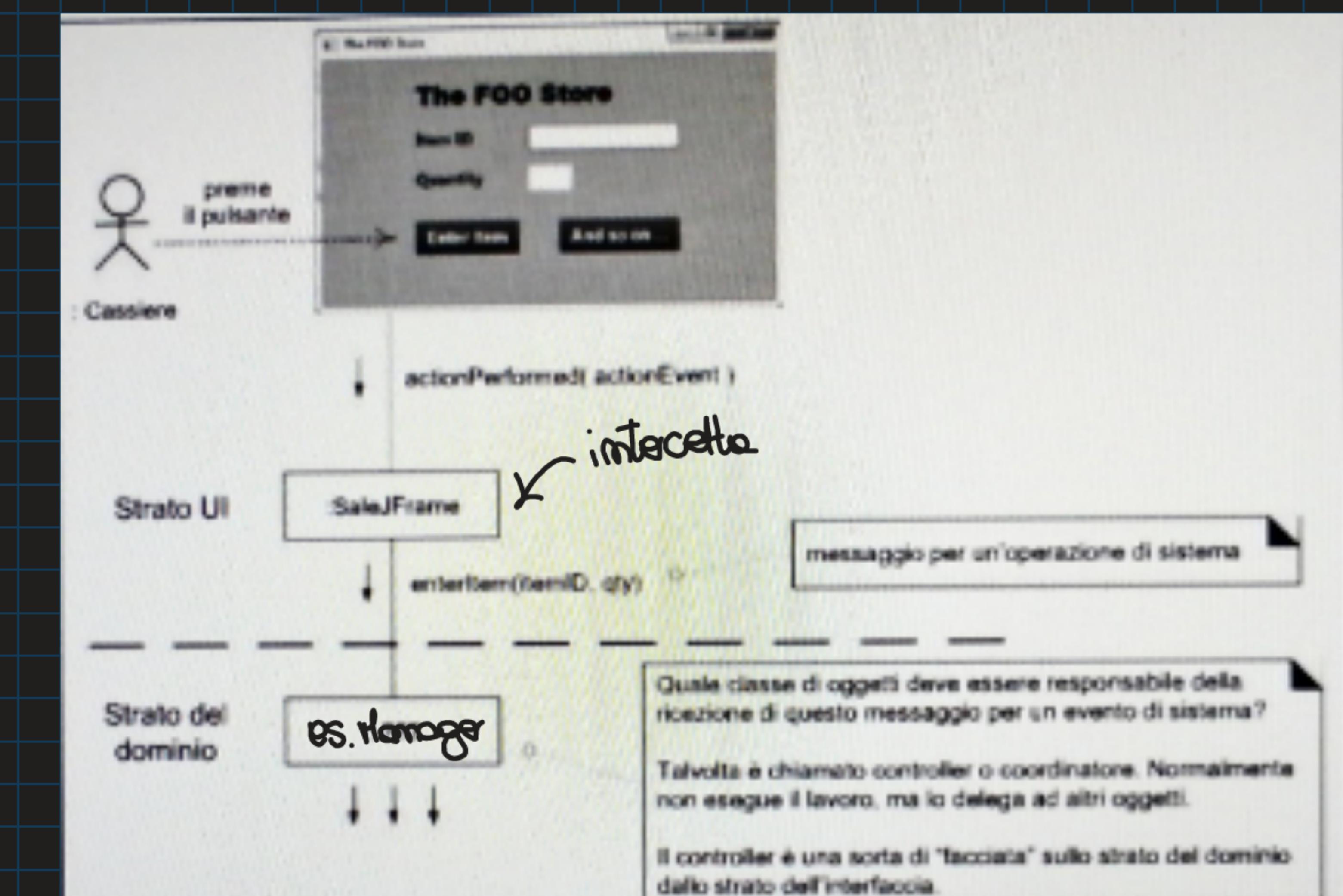
SOLUZIONE

Si decide un oggetto che rappresenta una di queste cose:

- è oggetto radice, rappresenta il "sistema" complessivo (variante grande controller)
- rappresenta uno scenario di un caso d'uso all'interno del (controllore di sessione o di caso d'uso) quale si verifica un'operazione di sistema

una facciata sopra agli altri strati dell'applicazione e fornisce un punto di accesso principale per le chiamate dei servizi del dominio attraverso gli altri strati sotstanti.

↳ controller diverso
per ogni caso d'uso.
Si sceglie quando
la prima soluzione
non è più a piacere con
coerenza bassa.



Definizione

sabato 1 luglio 2023 10:40

Potremo divisi in categorie:

1. CREATORIALI: **Abstract Factory e Singleton**: Risolvono problemi imprenti dell'inizializzazione, isolando l'initializzazione degli oggetti.
2. STRUTTURALI: **Adapter, Composite e Decorator**: Risolvono problemi imprenti la struttura delle classi e degli oggetti.
3. COMPORTAMENTALI: **Observer, State, Strategy e Visitor**: Formiscono soluzioni alle interazioni tra oggetti.

GRADIF Sono dei principi, i GOF sono schemi di progettazione avanzato.

↳ si preferisce la composizione rispetto all'ereditarietà tra classi

Nella composizione di oggetti le funzionalità sono ottenute **assemblando o componendo** gli oggetti per avere funzionalità più complesse, il riuso del software è ottenuto definendo una classe che usa altre classi (riuso black-box, i dettagli interni non sono conosciuti, infatti la classe B non è conosciuta alla classe A, la usa e basta).

Graficamente per capire:

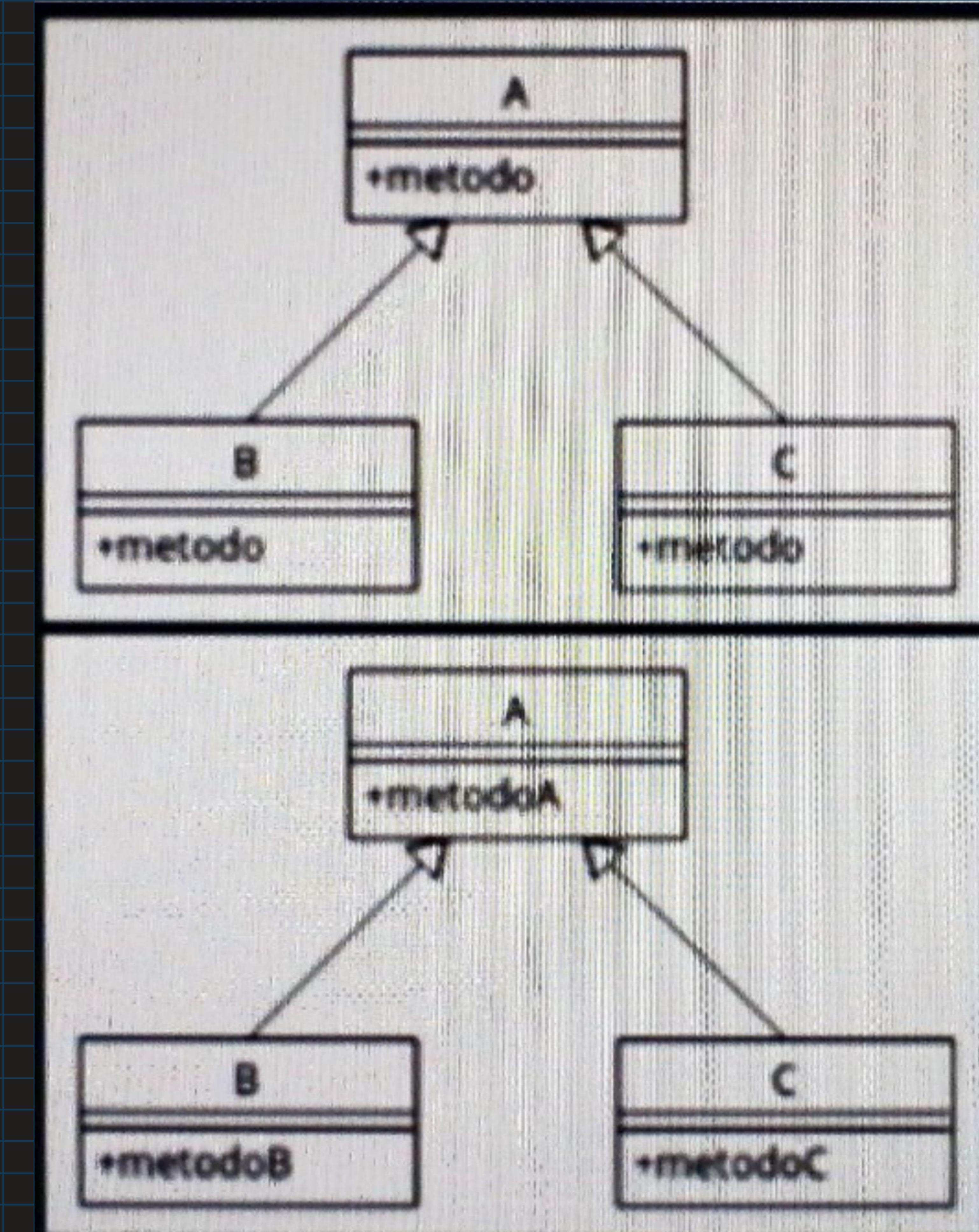
oggetti per avere funzionalità più complesse, il riuso del software è ottenuto definendo una classe che usa altre classi (riuso black-box, i dettagli interni non sono conosciuti, infatti la classe B non è conosciuta alla classe A, la usa e basta).

Inoltre L'ereditarietà di classi è definita staticamente, non è possibile cambiarla a tempo di esecuzione: se una classe estende un'altra, questa relazione è definita nel codice sorgente, non può cambiare a runtime.

Nei pattern GoF si può utilizzare l'ereditarietà per creare polimorfismo e non specializzazione

Si lavora con l'upcast

↳ Se sublassi, posso usare scambiando uno per l'altro, riduci i legami tra metodi, già esistente con lo stesso esempio, doppiazione senza doverlo fare



EREDITARIETÀ

SPECIALIZZAZIONE

Pattern Abstract Factory

sabato 1 luglio 2023 11:07

Creare istanze di oggetti che sono legati fra loro da diverse relazioni, devono quindi essere coordinati fra loro

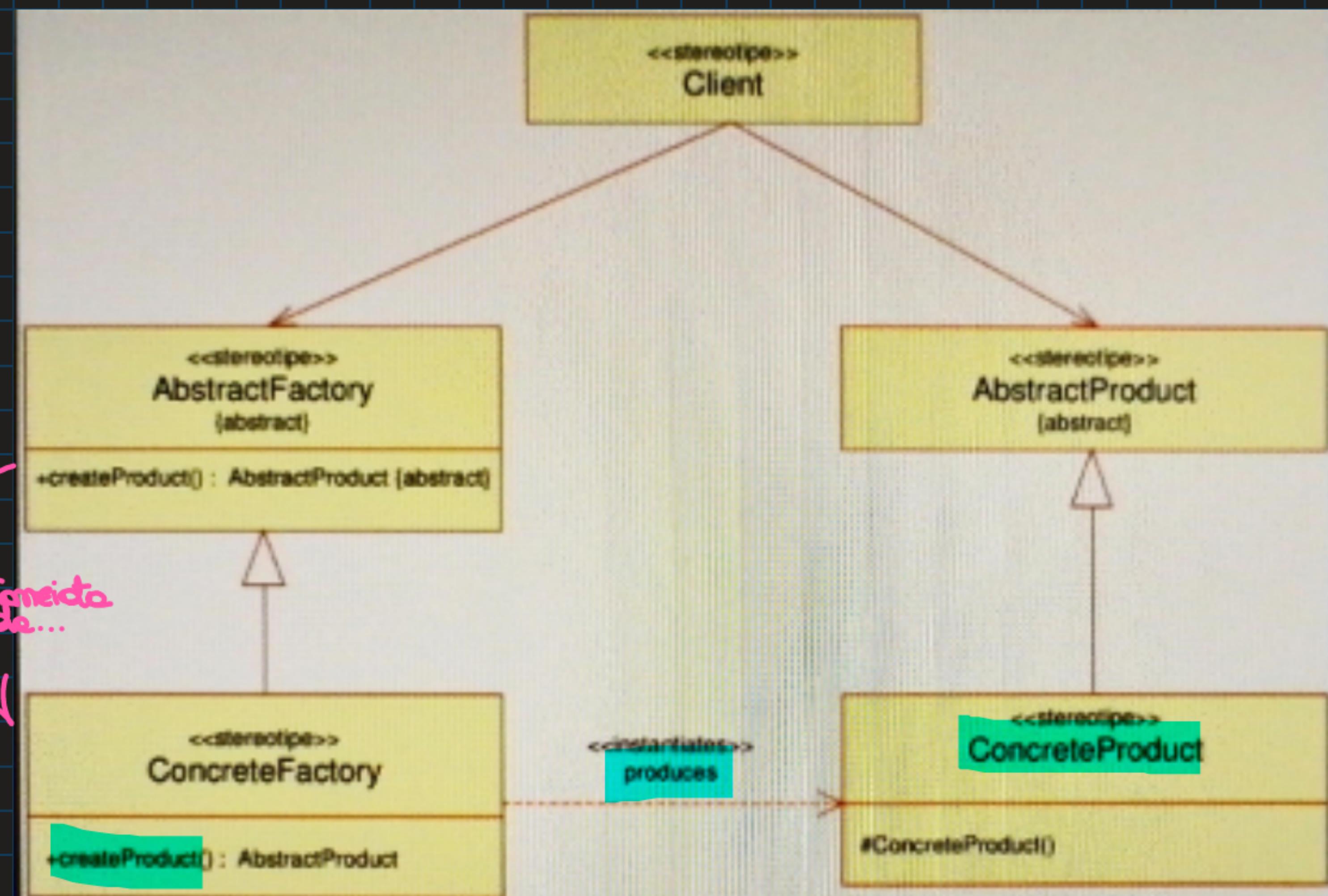
Abstract Factory

Nome: Abstract Factory

Problema: Come creare famiglie di classi correlate che implementano un'interfaccia comune?

Soluzione: Definire un'interfaccia factory (la factory astratta). Definire una classe factory concreta per ciascuna famiglia di elementi da creare. Opzionalmente, definire una vera classe astratta che implementa l'interfaccia factory e fornisce servizi comuni alle factory concrete che la estendono.

stereotype = interface, d'ora in poi si indica la classe specifica



PROBLEMA famiglie
Come creare classi correlate che implementano un'interfaccia comune?

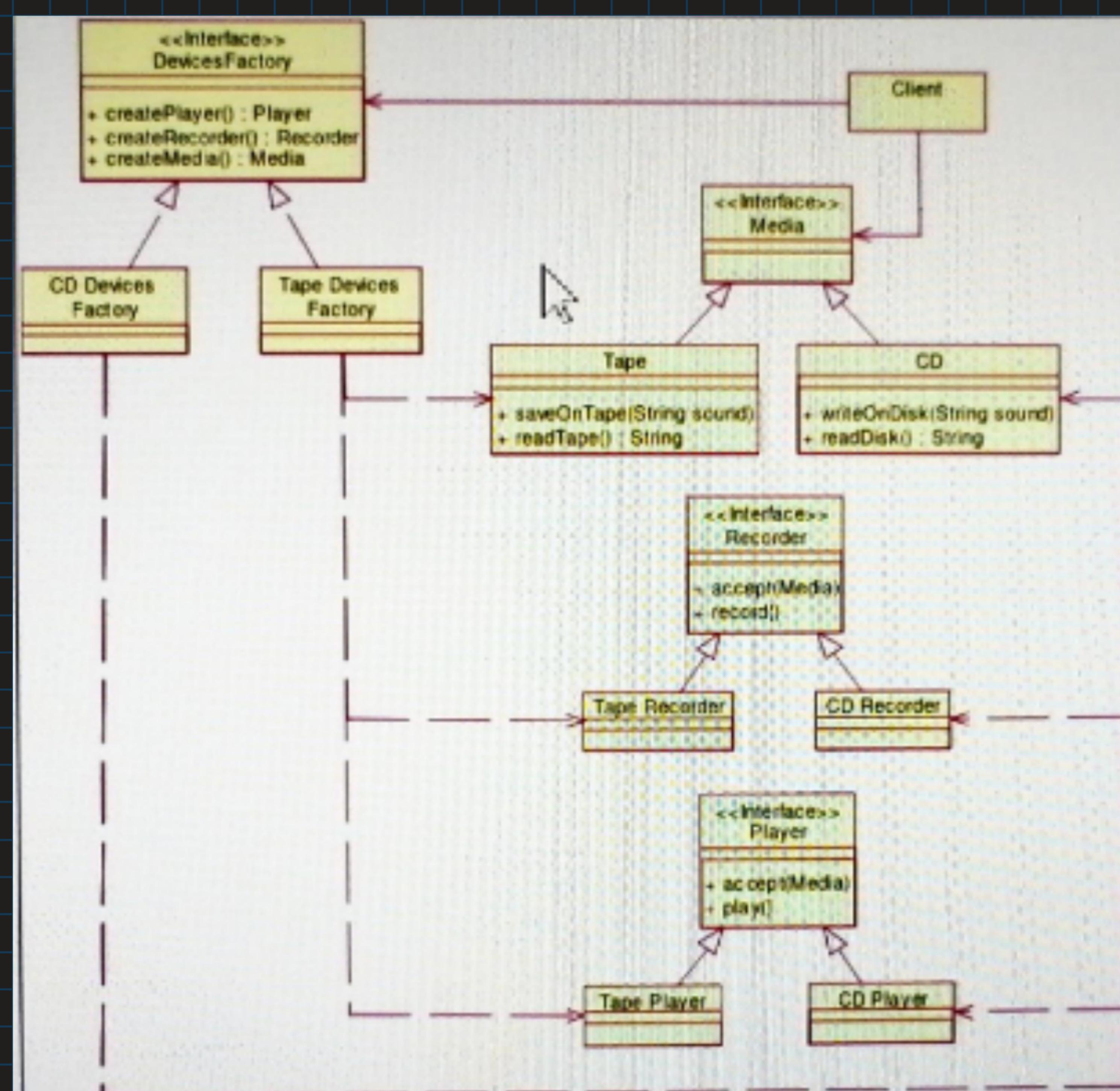
SOLUZIONE

Dogmiamo un'interfaccia factory, la classe che lo implementa può essere:

- concreta per ogni famiglia di elementi: da creare
- oppure
- astratta che fornisce servizi comuni alle sottoclassi concrete che lo estendono

```
public class AbstractFactoryExample {  
  
    public static void main ( String[] arg ) {  
  
        Client client = new Client();  
  
        System.out.println( "***Testing tape devices" );  
        client.selectTechnology( new TapeDevicesFactory() );  
        client.test( "I wanna hold your hand..." );  
  
        System.out.println( "***Testing CD devices" );  
        client.selectTechnology( new CDDevicesFactory() );  
        client.test( "Fly me to the moon..." );  
  
    }  
}
```

as. Una classe astratta ha due implementazioni
Concrete



Pattern Singleton

sabato 1 luglio 2023 11:34

Dare venire esattamente una sola istanza di una data classe.

Singleton

Nome: Singleton

Problema: È consentita (o richiesta) esattamente una sola istanza di una classe, ovvero un "singleton". Gli altri oggetti hanno bisogno di un punto di accesso globale e singolo a questo oggetto.

Soluzione: Difinisci un metodo statico (di classe) della classe che restituisce l'oggetto singleton.

Una volta creato l'oggetto singleton viene richieste alle classi che lo usano un oggetto, un puntatore, di oggetto singleton.

PROBLEMA

→ **SINGLETON**
È consentito una sola istanza per quella classe. Ci dà essere un singolo punto di accesso globale a questa istanza "singleton".

SOLUZIONE

Borimare un metodo statico della classe che restituisce l'oggetto singleton. Una classe sola viene incaricata di rispondere l'oggetto singleton e questo avrà anche un metodo che sarà l'unico accesso globale al nostro oggetto singleton.

Creiamo un'istanza e memorizzarla.

<<stereotype>> **Singleton**

-instance:Singleton

-Singleton()
+getNewInstance():Singleton

il costruttore è
privato chiede
un new

Se non esiste
altrimenti restituisce
quello creato

- Singleton come classe statica: si lavora con la classe statica, non un oggetto. Questa classe statica ha metodi statici che offrono i servizi richiesti, ad esempio:

```
public static class PrinterSpooler {  
    private PrinterSpooler() {  
        //  
    }  
    public static void print (String msg) {  
        System.out.println (msg);  
    }  
}
```

Quando il classloader carica la classe statica, quella è presente in un'unica copia.

- Singleton creato da un metodo statico che va a sostituire la new:

```
public static PrinterSpooler getinstance () {  
    if (instance==null) {  
        instance = new PrinterSpooler();  
    }  
    return instance;  
}  
  
public void print (String msg) {  
    System.out.println (msg);  
}
```

Pattern Adapter

sabato 1 luglio 2023 14:10

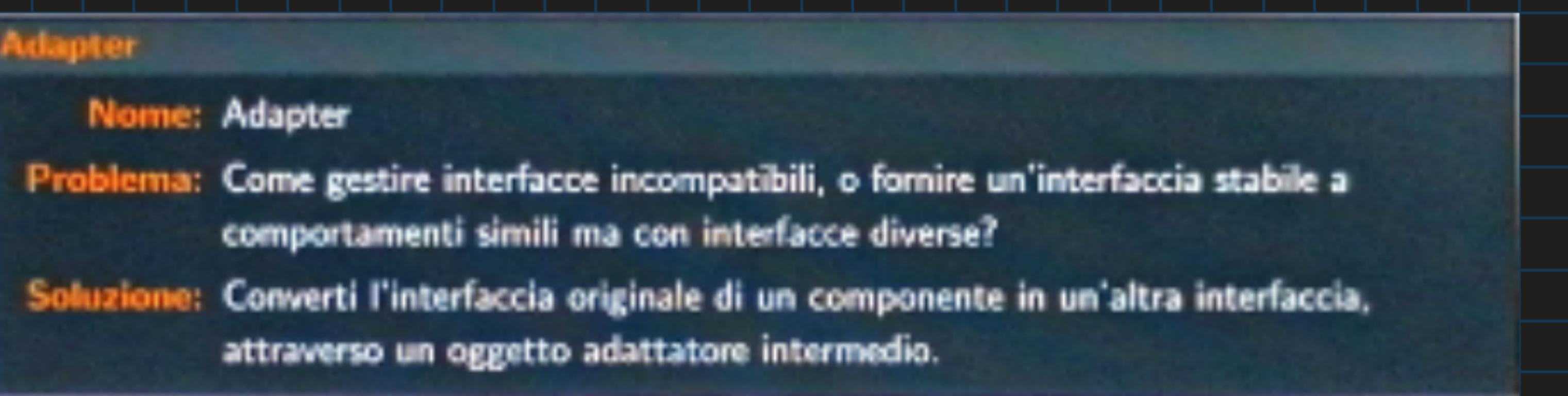
Vogliamo usare una classe che ha un'interfaccia con un'altra interfaccia.

Prendiamo in considerazione una coppia di oggetti software in una relazione client-server... per il servizio

Riceviamo una richiesta client di un servizio che deve essere dato da un altro server (lo so) solo che questo ha un'interfaccia diversa. Quindi il nostro scopo è convertire l'interfaccia incompatibile in un'altra compatibile tramite un adattatore che fa da INTERMEDIARIO

Ecco il funzionamento generale di Adapter:

1. In generale, un adattatore riceve richieste dai suoi client, per esempio da un oggetto dello strato di dominio, nel formato client.
2. L'adattatore poi adatta, trasforma, una richiesta ricevuta in una richiesta nel formato del server.
3. L'adattatore invia la richiesta al server.
4. Se il server fornisce una risposta, lo fa nel formato del server.
5. L'adattatore adatta, trasforma, la risposta ricevuta dal server in una risposta nel formato del client e poi la restituisce al suo client.



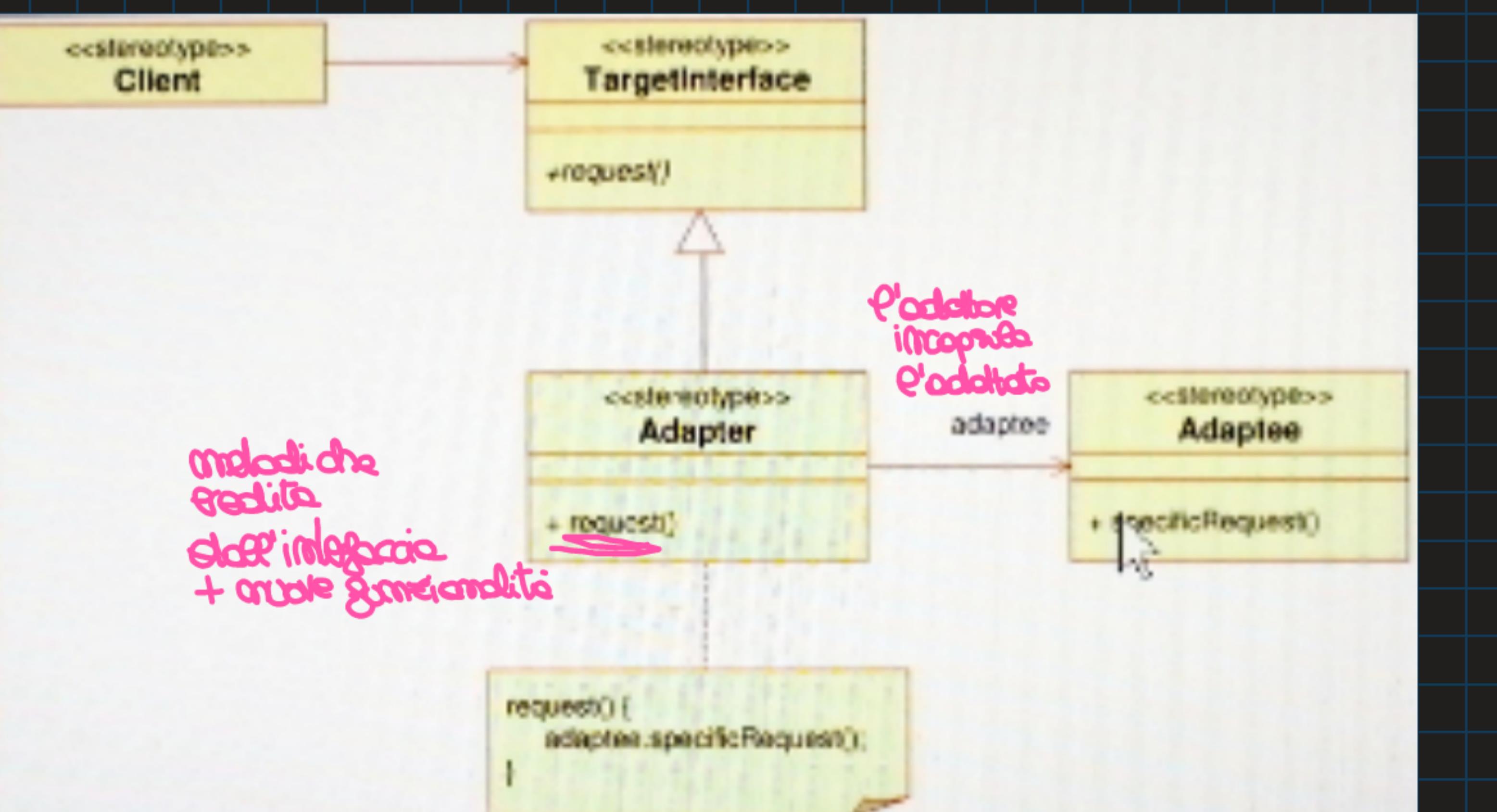
PROBLEMA

Come gestire interfacce incompatibili? Come fornire un'interfaccia stabile comune per uscire servizi diversi con interfacce diverse?

Se cliente (chiede molto specifico un trisangolo)

ADATTATORE

INTERFACCIA



metodi che
scrive
sull'interfaccia
+ creare genericità

P'adattore
incarna
l'adattato

adaptee

Adaptee

+ specificRequest()

metodo
nella
adattatore

specific request

```
public class RectangleObjectAdapter implements Polygon {
    Rectangle adaptee;
    private String name = "NO NAME";

    public RectangleObjectAdapter() {
        adaptee = new Rectangle();
    }

    public void define( float x0, float y0, float x1, float y1,
                       String col ) {
        float a = x1 - x0;
        float l = y1 - y0;
        adaptee.setShape( x0, y0, a, l, col );
    }
}
```

Pattern Composite

sabato 1 luglio 2023 15:16

composite

Nome: Composite

Problema: Come trattare un gruppo o una struttura composta di oggetti (polimorficamente) dello stesso tipo nello stesso modo di un oggetto non composto (atomico)?

Soluzione: Definisci le classi per gli oggetti composti e atomici in modo che implementino la stessa interfaccia

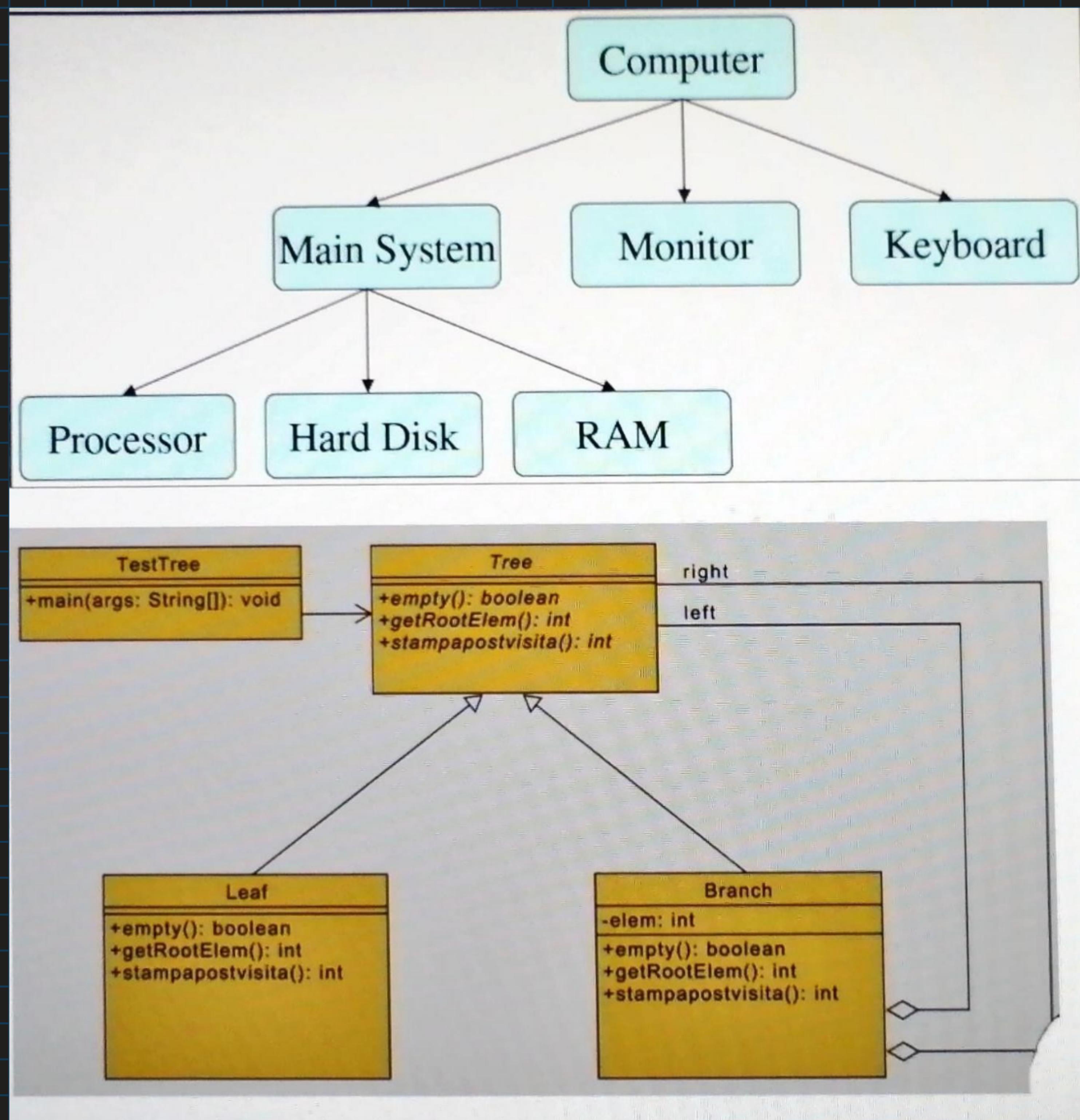
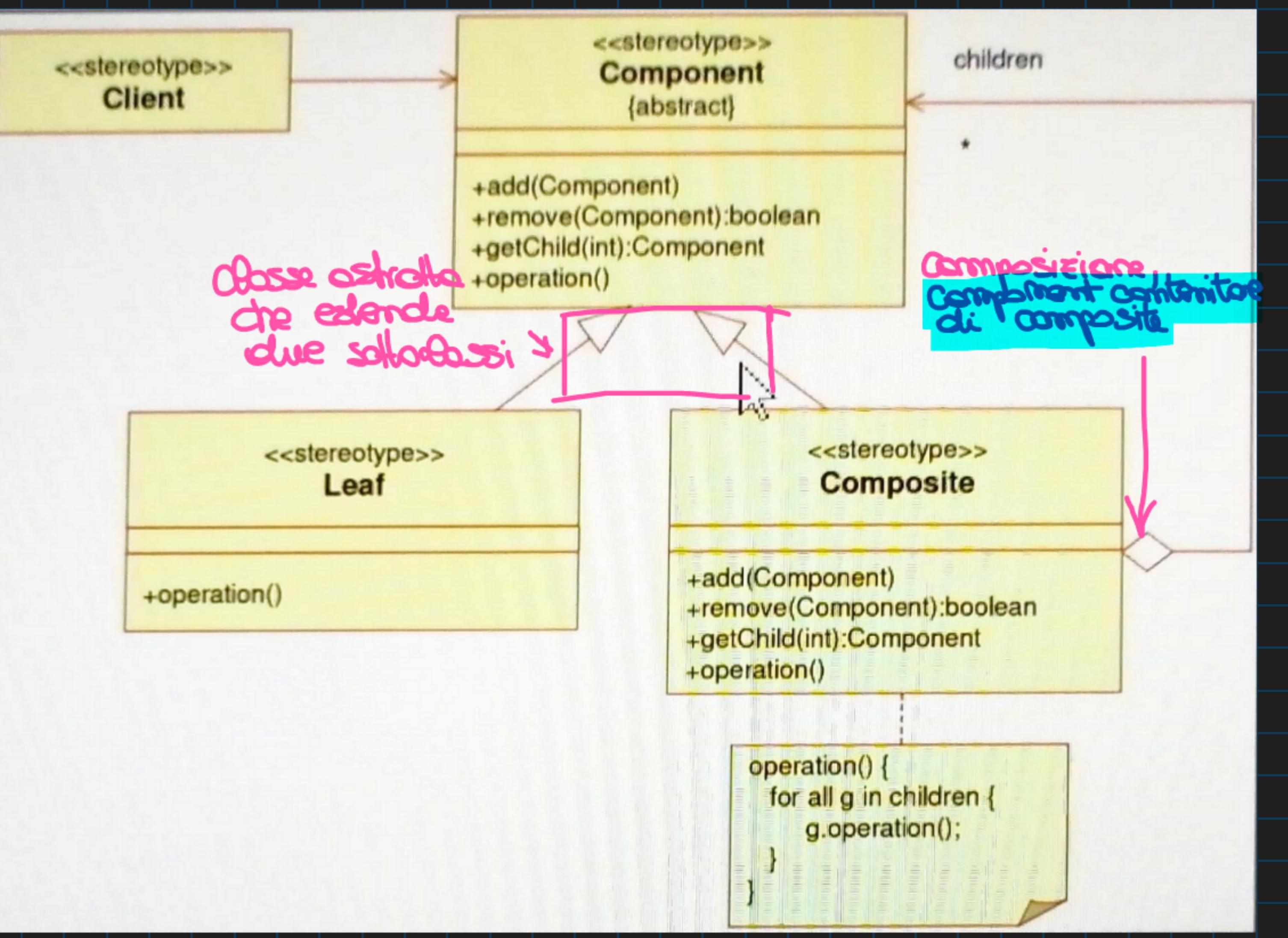
Questo pattern permette di costruire strutture ricorsive in modo che per l'utente è infuso. Struttura viene vista come una semplice entità.

PROBLEMA

Come trattare una struttura composta di oggetti polimorfi dello stesso tipo nello stesso modo in cui trattiamo un oggetto "non composto", non composto?

SOLUZIONE

Implementiamo la stessa interfaccia per i tipi di oggetti composti e atomici.



Pattern Decorator

sabato 1 luglio 2023 15:33

Decorator

Nome: Decorator

Problema: Come permettere di assegnare una o più responsabilità addizionali ad un oggetto in maniera dinamica ed evitare il problema della relazione statica? Come provvedere una alternativa più flessibile al meccanismo di sottoclasse ed evitare il problema di avere una gerarchia di classi complessa?

Soluzione: Inglobare l'oggetto all'interno di un altro che aggiunge le nuove funzionalità.

Vogliamo aggiungere nuove responsabilità durante le ~~creazioni~~ del sistema, questo dinamicamente.
Quando vogliamo aggiungere una proprietà estendiamo una classe con un'altra, solo che ~~questo~~ viene e' un'operazione statica perché lo facciamo dall'inizio quando scriviamo il codice.
Aggiungiamo una nuova responsabilità dinamicamente intercalando l'oggetto a cui vogliamo aggiungere responsabilità ~~intorno~~ un altro che offre aggiunte

Problema

Assegnamenti dinamici di una o più responsabilità ad un oggetto, come evitare lo statico? Quindi evitare sottoclassi dell'oggetto

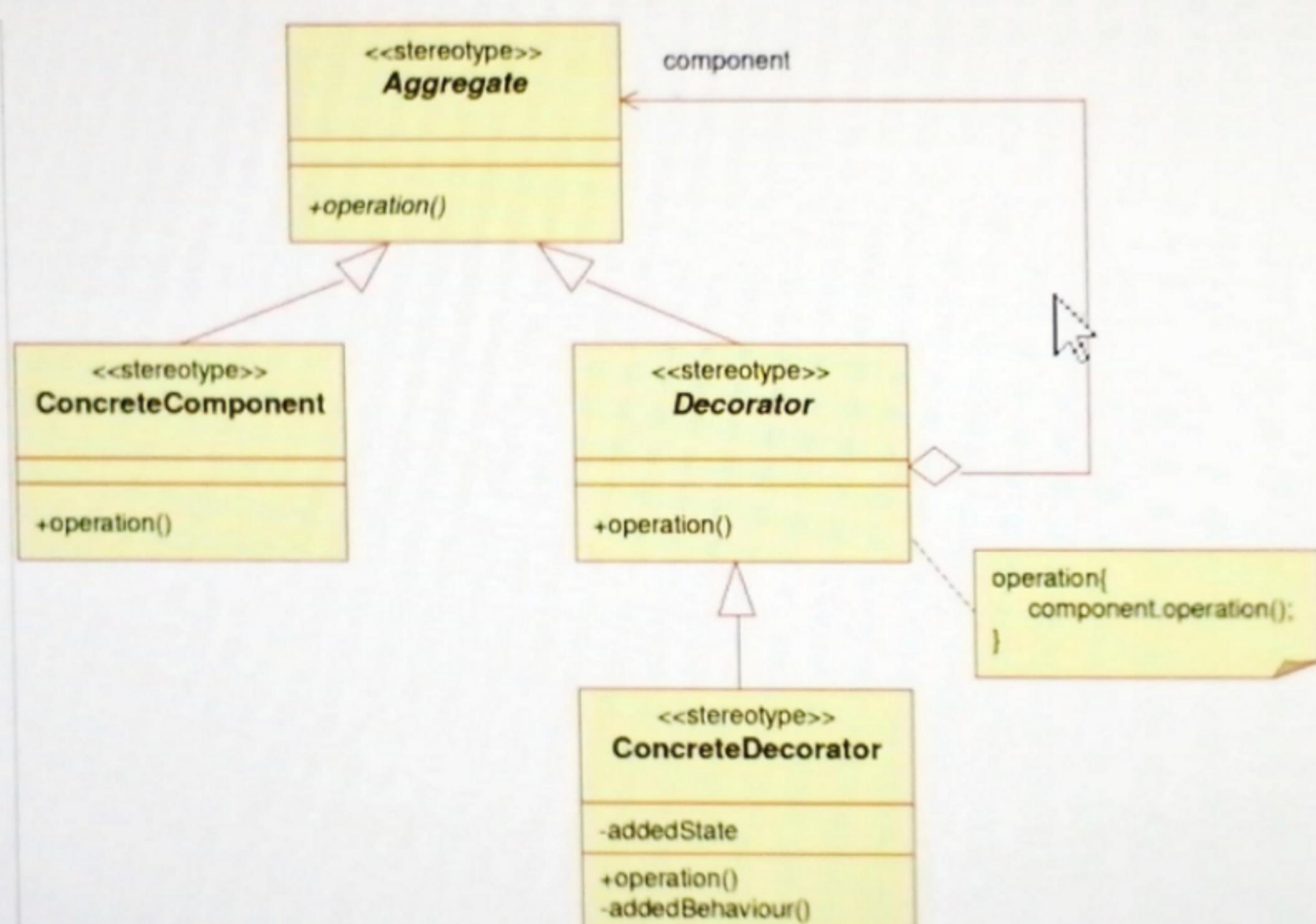
Soluzione

Immaginare è' oggetto in un altro che glielo aggiunge

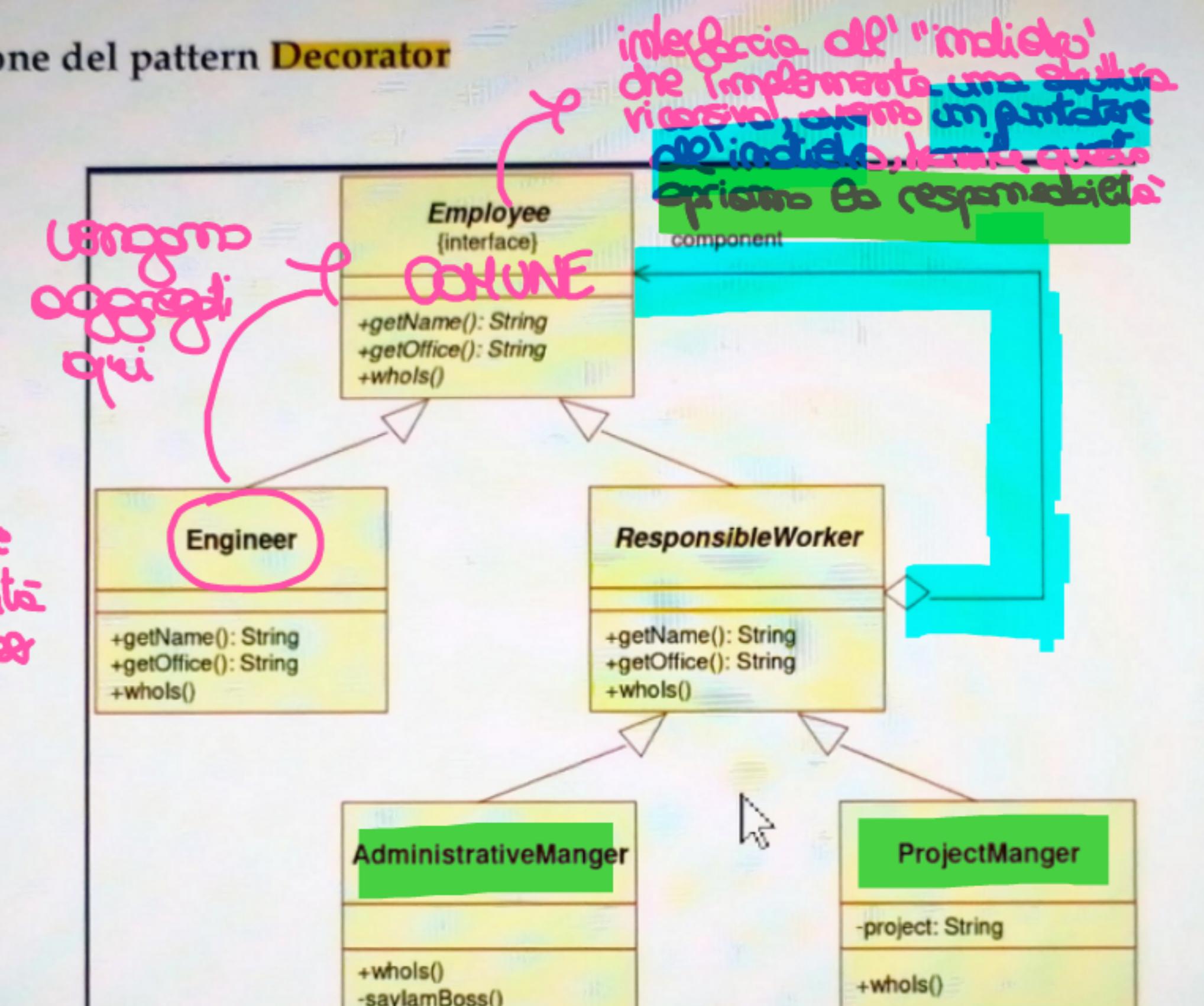
Decorator permette quindi di:

- Permette di aggiungere responsabilità ad oggetti individualmente, dinamicamente e in modo trasparente, ossia senza impatti sugli altri oggetti.
- Le responsabilità possono essere ritirate. 
Ovvero butto via il puntatore a quello del wrapper e uso di nuovo quello precedente.
- Permette di evitare l'esplosione di sottoclassi che si ha quando estendiamo classi e creiamo gerarchie con extends.

Struttura del pattern **Decorator**

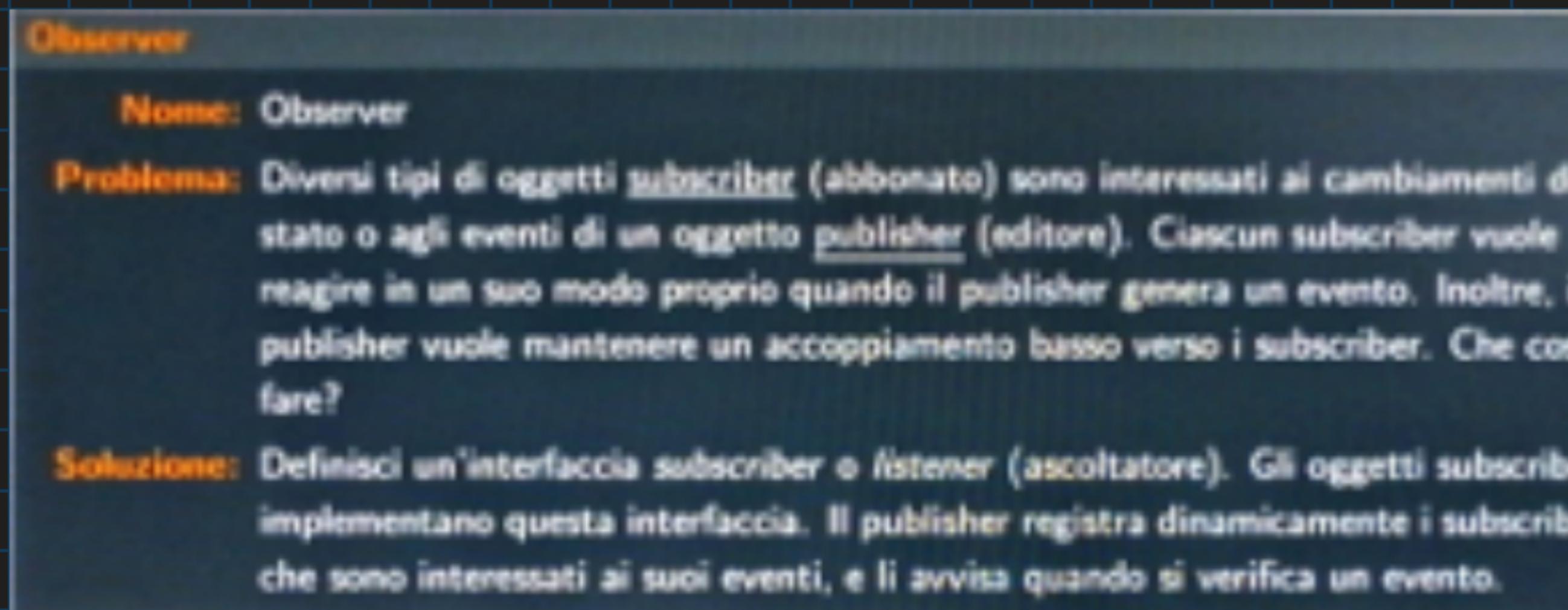


Applicazione del pattern **Decorator**



Pattern Observer

sabato 1 luglio 2023 16:34



E' ad esempio il modo con cui si implementano i listener nelle interfacce grafiche; abbiamo un generatore di eventi e una lista di osservatori, quando il generatore cambia stato notifica tutti i suoi osservatori.

Questi osservatori interessati al cambiamento eseguono poi operazioni in risposta alla notifica. Observer definisce una dipendenza tra oggetti di tipo uno-a-molti: quando lo stato di un oggetto cambia, tale evento viene notificato a tutti gli oggetti dipendenti, essi vengono automaticamente aggiornati.

L'oggetto che notifica il cambiamento di stato non fa alcuna assunzione sulla natura degli oggetti notificati: le due tipologie di oggetti sono disaccoppiati.

Il numero degli oggetti affetti dal cambiamento di stato di un oggetto non è noto a priori. Fornisce un modo per accoppiare in maniera debole degli oggetti che devono comunicare (eventi).

I publisher conoscono i subscriber solo attraverso un'interfaccia, e i subscriber possono registrarsi (o cancellare la propria registrazione) dinamicamente con il publisher.

Problema

Esempio: diversi tipi di abbonati sono interessati ai cambiamenti di stato o agli eventi di un editore e ogni abbonato vuole reagire a modo suo all'evento generato dall'editore.

ABBRONATO = SUBSCRIBER, EDITORE = PUBLISHER

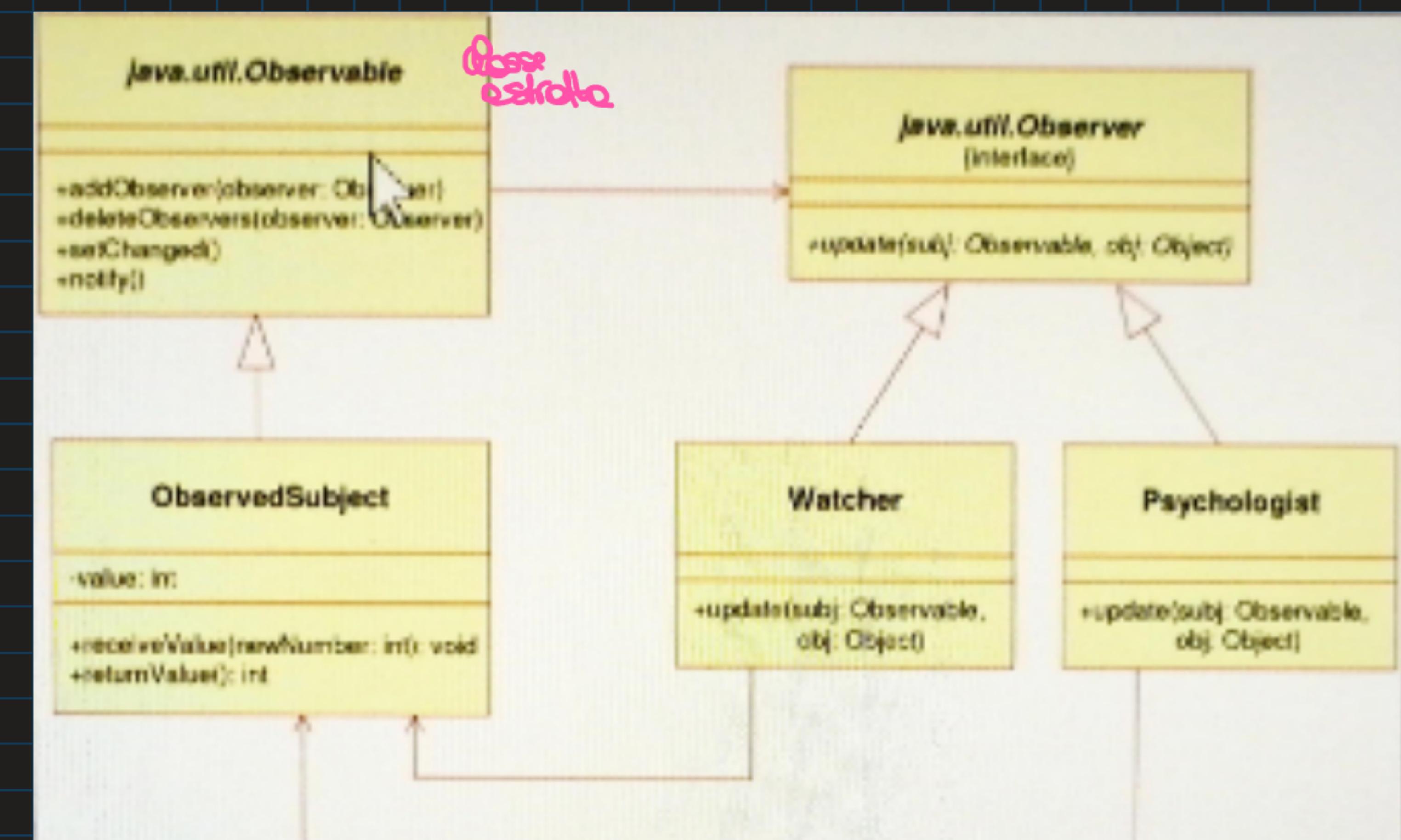
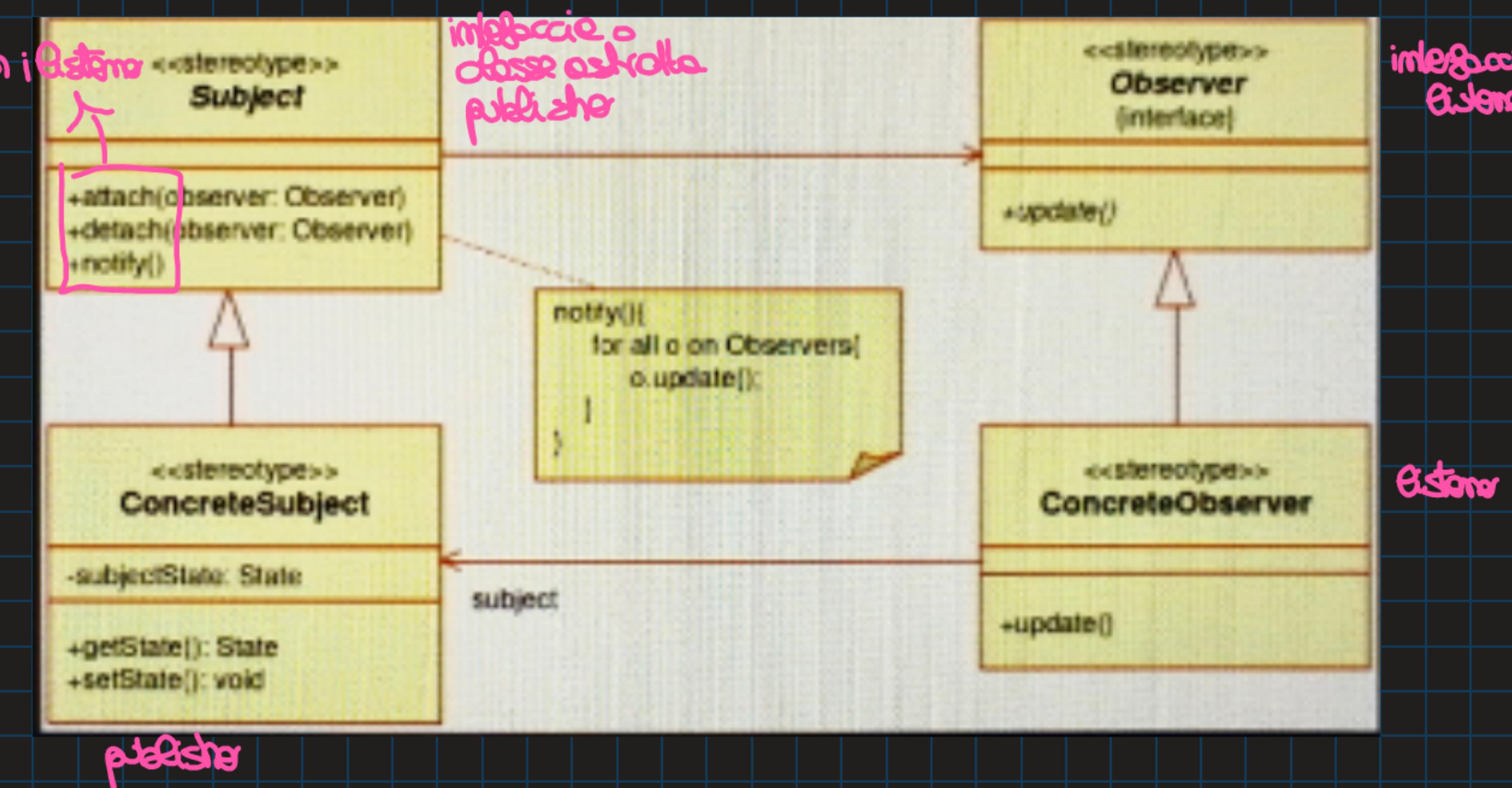
Come possono più oggetti subscriber reagire a modo suo all'evento generato da publisher mantenendo comunque un accoppiamento basso?

Soluzione

ascoltatore
definiamo un'interfaccia **subscriber** o **listener**. Gli oggetti ascoltatori implementano questa interfaccia. Il publisher registra dinamicamente gli ascoltatori interessati ai suoi eventi e li avvisa.

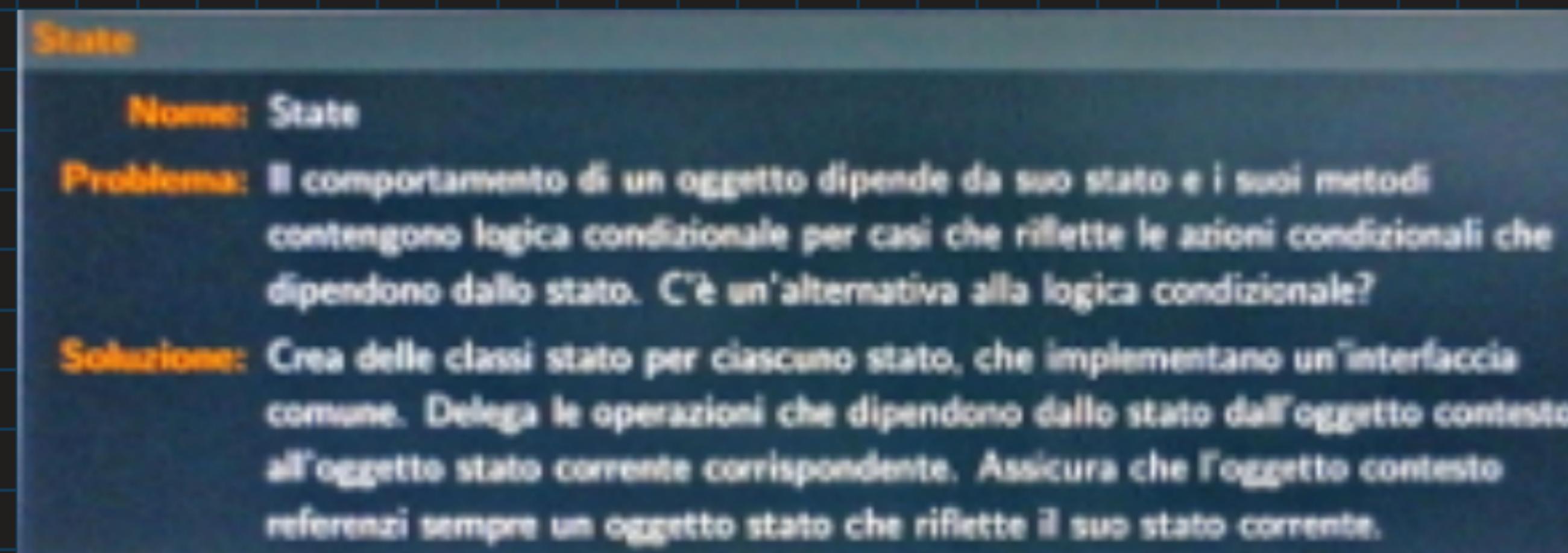
→ ascoltatori: soluzione degli interfacce

come si
solva le
sue istanze



Pattern State

sabato 1 luglio 2023 11:47



Comportamento in base allo stato:

es) Dato un orologio, l'ora corrisponde ad uno stato diverso e in base a che ora siamo abbiamo metodi diversi.

Dobbiamo creare una classe per ogni stato con un'interfaccia comune.

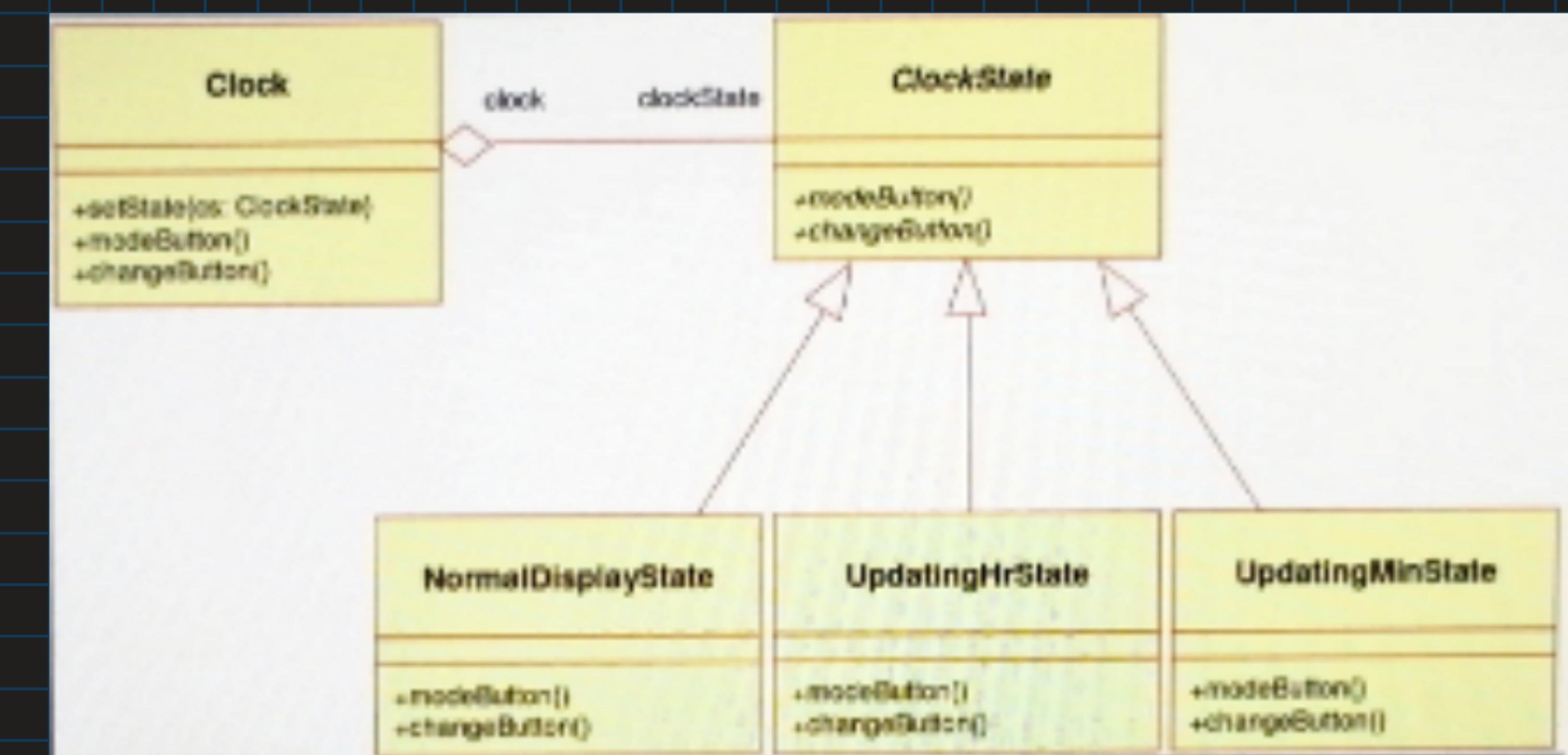
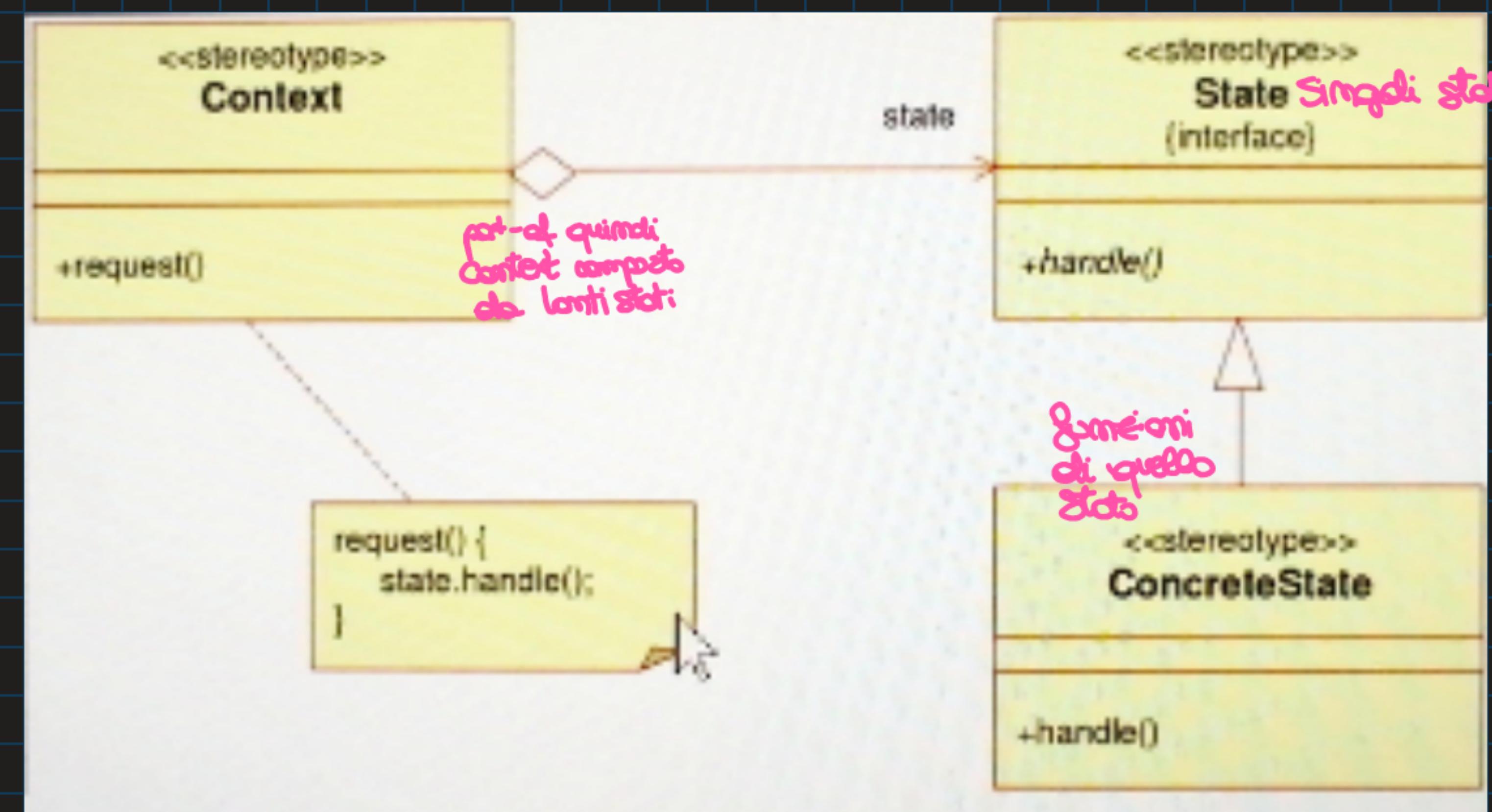
Problema

Il comportamento di un oggetto dipende dal suo stato. I metodi implementano la logica condizionale, ogni azione, comunque, dipende dallo stato. Vogliamo un'alternativa alla logica condizionale.

(es. se accade questo, fa quello)

Soluzione

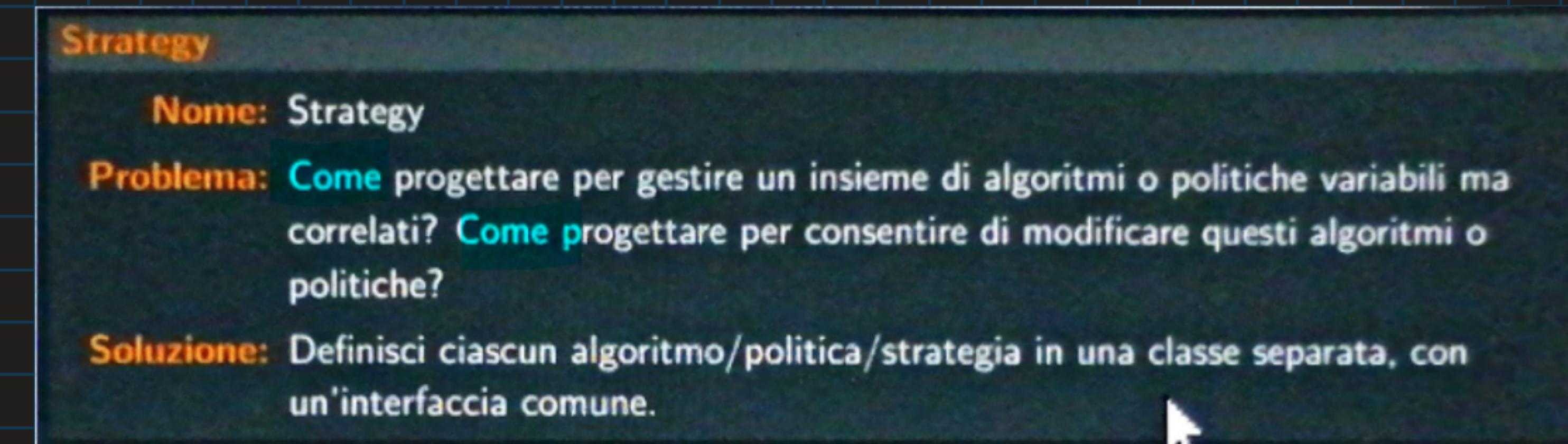
Una classe stato per ogni stato che implementano un'interfaccia comune. Ogni stato ha le operazioni che gli dipendono implementate nella classe dello stato in questione.



Pattern strategy

venerdì 30 giugno 2023 10:25

Ogni pattern ha un suo problema ed una soluzione



Soluzione proposta dal pattern: Supponiamo di avere un determinato **algoritmo di ordinamento**, e supponiamo di volerlo cambiare senza dover cambiare il codice. (Nota che l'obiettivo finale rimane sempre lo stesso ovvero ordinare la struttura dati).

La soluzione è quella di mettere questo algoritmo in una classe, creo un'istanza di questa classe in cui metto all'interno l'algoritmo di ordinamento in un metodo e passo l'oggetto creato al metodo che vuole ordinare.

L'oggetto contesto è l'oggetto a cui va applicato l'algoritmo.

↳ **contesto associato a strategia che è quello che implementa l'algoritmo**

L'oggetto contesto è associato ad un **oggetto strategia**, questo oggetto strategia è l'oggetto che implementa un algoritmo.

Il **pattern strategy** ha le seguenti caratteristiche:

- permette di definire una famiglia di algoritmi e li incapsula ognuno e li rende intercambiabili.

Prende questi algoritmi e li mette dentro a delle classi, queste sono correlate tra di loro e hanno una interfaccia comune, ovvero sono tutte sottoclassi di questa interfaccia comune.

L'intercambiabilità è permessa grazie al **binding dinamico**.

- Permette di modificare gli algoritmi in modo indipendente dai clienti che fanno uso di essi.
- Disaccoppia gli algoritmi dai clienti che vogliono usarli dinamicamente.
- Permette che un oggetto client possa usare indifferentemente uno o l'altro algoritmo
- Usa la composizione invece dell'ereditarietà: i comportamenti di una classe non dovrebbero essere ereditati ma piuttosto incapsulati usando la dichiarazione di interfaccia

→ poiché vengono usate interfacce al posto che classi già esistenti, poiché è redditizio fare questo

il caso che vuoi risolvere

PROBLEMA

più algoritmi o politiche diverse → es. obiettivi comuni →
fra loro ma correlati (simili)
come oggettivi e ambig. così

SOLUZIONE

classi separate per ogni algoritmo/politica
una interfaccia comune

classi ordinamento ↗ CONTESTO

classi ordinamento ↗ costruttore (classi del tipo ordinamento: es. sort)

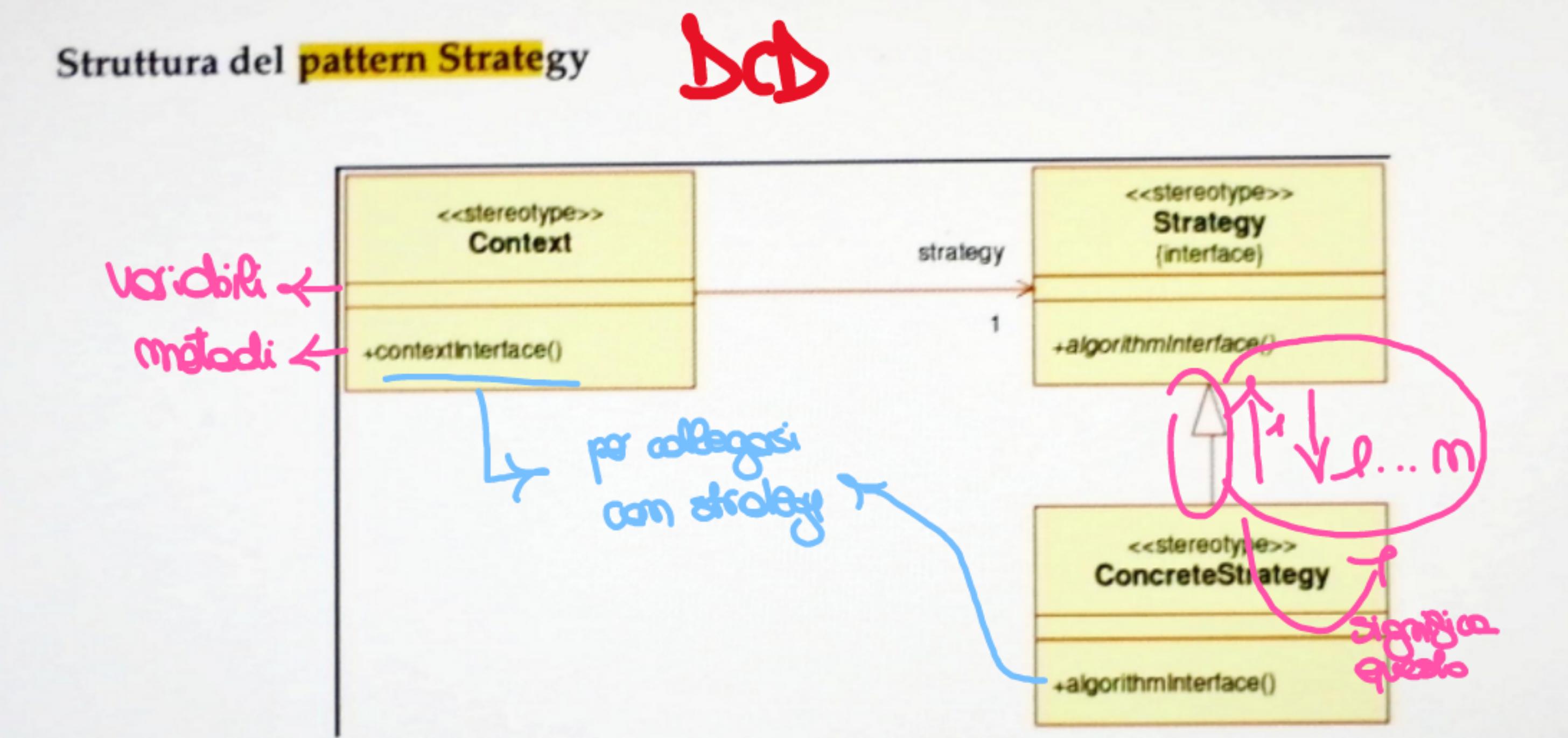
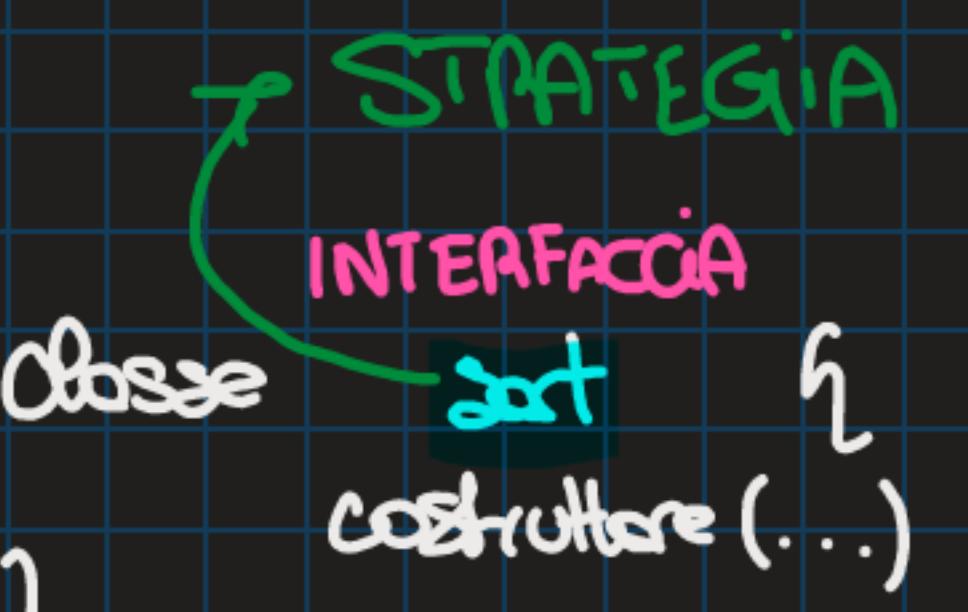
}

CODICE PRINCIPALE

sort s = new quickSort();
ordinamento o = new ordinamento(s);

Così ordiniamo nel modo in cui vogliamo senza compiere il codice delle interfacce e ordinamento, se vogliamo cambiare ordinamento basta creare nuove variabili di sort

obiettivi comuni. Prendiamo ad esempio il processo di pagamento con carta di credito è diverso rispetto al processo di pagamento con contanti, tuttavia soddisfa il comune obiettivo di effettuare il pagamento.



Pattern visitor

venerdì 30 giugno 2023 11:08

Visitor

Nome: Visitor

Problema: Come separare l'operazione applicata su un contenitore complesso dalla struttura dati cui è applicata? Come poter aggiungere nuove operazioni e comportamenti senza dover modificare la struttura stessa? Come attraversare il contenitore complesso i cui elementi sono eterogenei applicando azioni dipendenti dal tipo degli elementi?

Soluzione: Creare un oggetto (ConcreteVisitor) che è in grado di percorrere la collezione, e di applicare un metodo proprio su ogni oggetto (Element) visitato nella collezione (avendo un riferimento a questi ultimi come parametro). Ogni oggetto della collezione aderisce ad un'interfaccia (Visitable), che consente al ConcreteVisitor di essere accettato da parte di ogni Element. Il Visitor analizza il tipo di oggetto ricevuto, fa l'invocazione alla particolare operazione che deve eseguire.

Ogni visitor ha le funzioni per ogni tipo esatto di elemento.

Così per lo stampa che ha solo l'operazione dello stampa, però più funzioni quanti sono i tipi esatti di elementi

PROBLEMA → aggiungere
separare l'operazione su un contenitore di
una struttura.

Vogliamo distinguere le operazioni di ogni
contenitore, o seguenti: fra di loro.

Oppure applicare la stessa funzione a tipi diversi
di elementi nel contenitore

SOLUZIONE

L'oggetto **visitor**, visita tutti gli elementi, di qualsiasi tipo siano,
della struttura e applica ad ognuno di essi l'operazione desiderata.

Ogni elemento deve prima di tutto aderire all'interfaccia **visitable**
che consente al visitor di essere accettato.

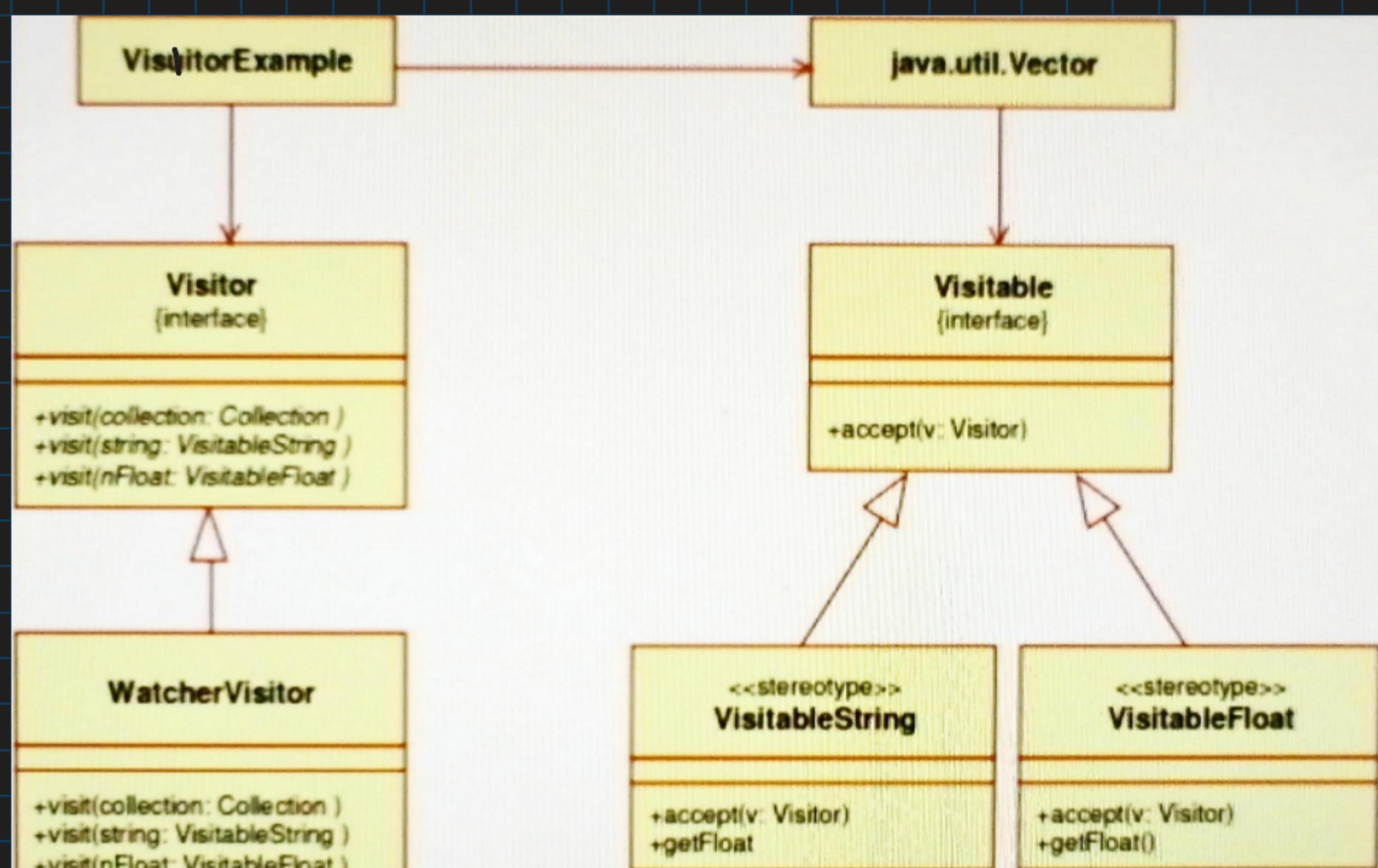
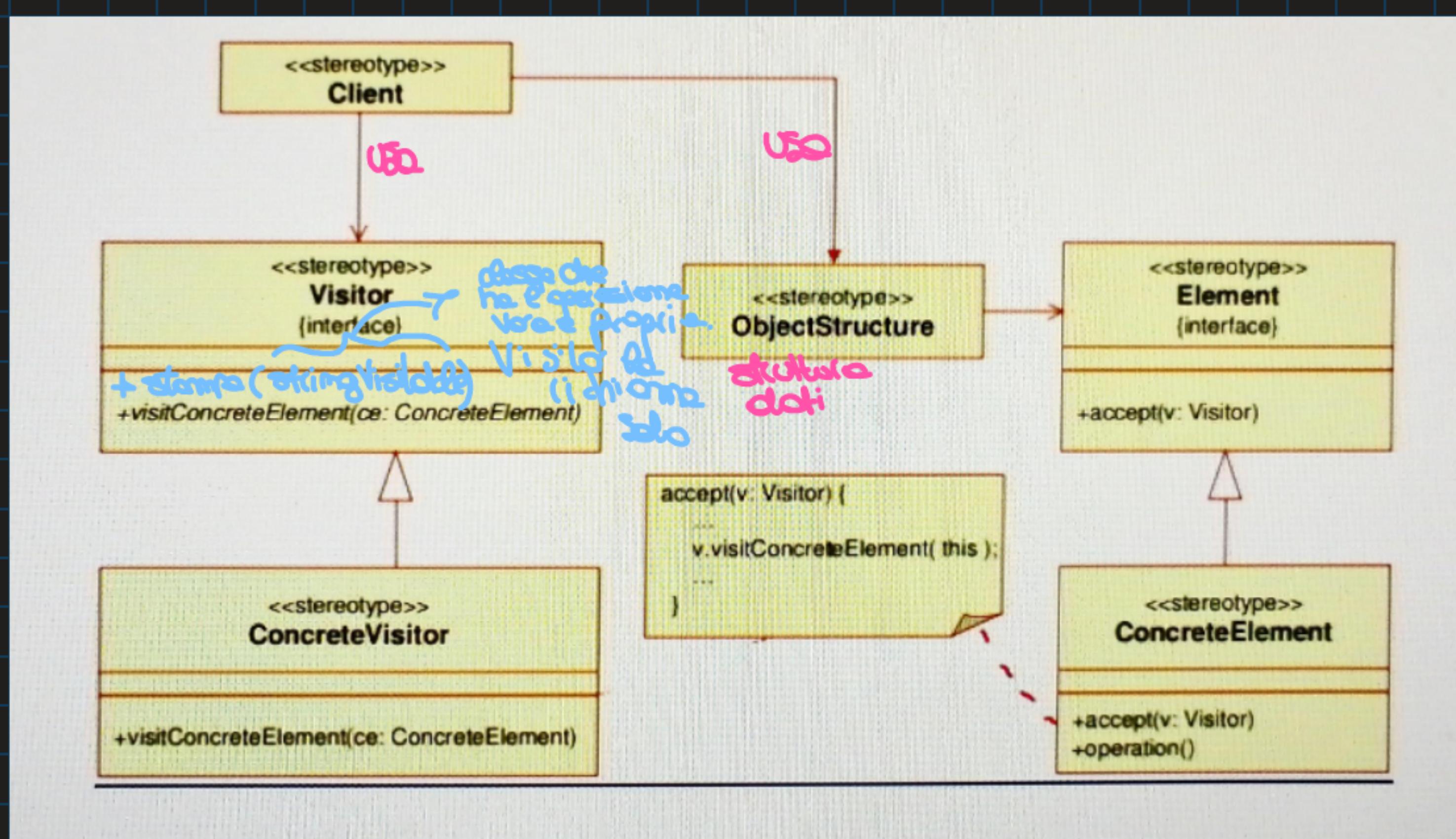
es. struttura dati = array
contenitore = ~~celle array con un valore fisso~~
~~ad esempio un puntatore mappa celle~~

Per capire, qua il problema è: supponiamo di avere una struttura dati (es. un array che contiene puntatori a oggetti di tipo esatto diverso ma con tipo apparente comune).

Su tutti dobbiamo fare una operazione (ad esempio stampa, quindi questa è una funzione che noi vogliamo passare alla struttura in modo tale che questa la applichi).

class → class associata
class → class polimerica
↓
implementa l'interfaccia ←
lo stesso

Ogni classe avrà le metodi accept che accetta il visitatore,
questa funzione fa partire il visitatore che andrà alemento
e applica la funzione ad esso desiderata.



ES IMPLEMENTAZIONI JAVA

```
public static void main(String[] args) {

    int n = 10;
    Lista l = new ListaVuota();
    for (int i = 0; i < n; i++)
        l.aggungi(i);

    Visitatore v1 = new Stampa();
    Visitatore v2 = new Somma();
    l.accetta(v1);
    l.accetta(v2);
    System.out.println("Somma: " + ((Somma)v2).getSomma());
}
```