

Lez 27 (27/03)

Vediamo un caso di studio relativo alla progettazione di un software con architettura Service Oriented.

Parliamo di un software da utilizzare per fare online shopping, partendo da un insieme di requisiti astratti, analisi dei requisiti (dove vediamo un approccio leggermente diverso da quello visto nella prima parte del corso, sottolineando che non esiste un metodo migliore) e poi progettazione effettiva.

Web-based Online Shopping System

- In the **Web-based Online Shopping System**, customers can request to purchase one or more items from the supplier
- The customer provides personal details, such as address and credit card information
- This information is stored in a customer account
- If the credit card is valid, then a delivery order is created and sent to the supplier
- The supplier checks the available inventory, confirms the order, and enters a planned shipping date
- When the order is shipped, the customer is notified, and the customer's credit card account is charged

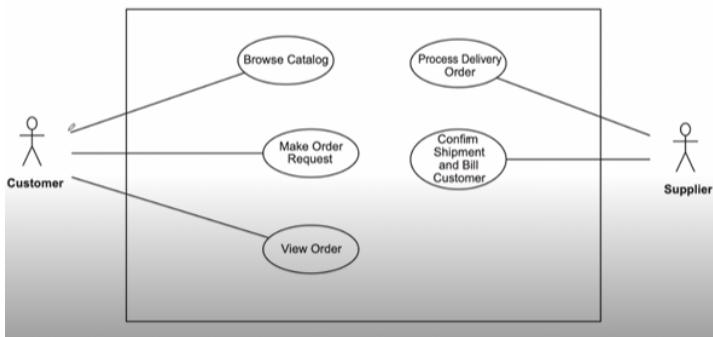
Il cliente fornisce dettagli personali, memorizzati nel customer account. Se il cliente effettua un ordine e la carta di credito è valida, un ordine di consegna è inviato al fornitore che controlla che il prodotto sia presente nell'inventario, in caso positivo conferma l'ordine e dice al cliente la data consegna prevista. Quando l'ordine è in consegna il cliente è notificato e la carta di credito addebitata.

A partire da questo insieme di requisiti li analizzeremo sia dal punto di vista comportamentale che strutturale e in questo caso invece che partire dal modello dei dati partiamo dal modello comportamentale (in particolare dalla modellazione dei casi d'uso). *Terminata l'analisi dei requisiti passeremo alla fase di progettazione, dove introdurremo in modo specifico la caratterizzazione service-oriented dell'applicazione da realizzare.*

Partendo dal modello comportamentale, essendo in assenza di un iniziale class diagram, ciò che faremo è identificare i casi d'uso e per ciascun caso d'uso descriverne il funzionamento facendo uso degli activity diagram.

Passeremo poi al modello dei dati per definire effettivamente le classi per poi tornare al comportamentale per raffinare in particolare l'object interaction, usando stavolta i communication diagram invece che i sequence diagrams come avevamo fatto nel primo modulo.

Use Case Diagram



Partiamo quindi dalla modellazione dei casi d'uso e dal relativo diagramma. Si identificano due attori principali che interagiscono con il software: il cliente e il fornitore.

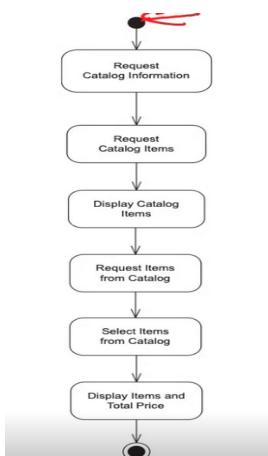
Ricordando che ogni caso d'uso deve essere a funzionalità completa (include anche eventuali flussi alternativi etc..), **indipendente dagli altri** e **identificato come tale solo se attivato da almeno un attore**, per identificarli lato cliente mi chiedo come il cliente interagisce con il software.

Tre casi d'uso: Sfoglia il catalogo di prodotti, può sottomettere un ordine e se ha sottomesso un ordine può visualizzarne lo stato.

Lato fornitore invece: processare l'ordine richiesto dall'utente (per verificare disponibilità inventario, carta di credito etc..) e confermare la spedizione e addebitare la carta del cliente.

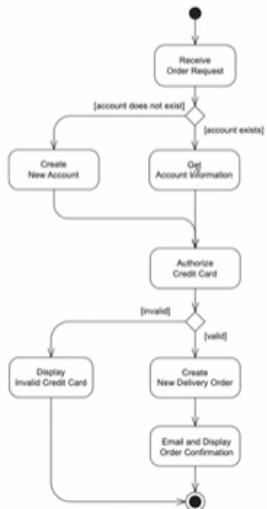
Sappiamo che i diagrammi di casi d'uso sono utilizzabili a diversi livelli di astrazione, qui alto livello e infatti mancano le frecce tra le associazioni (implicite).

A questo punto per ogni caso d'uso forniamo la relativa specifica usando il diagramma delle attività anche in assenza di class diagram (ricorda come nel primo modulo si partiva dai requisiti, che sono frasi nominali ossia dove il sostantivo prevale sulla parte verbale, e si cercavano di classificare i sostantivi come classi candidate o meno mentre in questo caso approccio diverso!).



Nell'activity diagram come sappiamo nodo iniziale e finale, l'iniziale corrisponde all'evento di attivazione da parte dell'attore e inizia la specifica di caso d'uso per portarlo a buon fine: nell'esempio la specifica del caso d'uso per sfogliare il catalogo di prodotti; dalla richiesta del catalogo e i suoi item al mostrarli a schermo con il relativo prezzo.

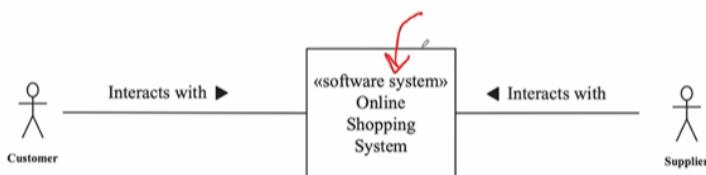
Facciamo la specifica per ognuno dei casi d'uso individuati



In questo caso per il caso d'uso di richiesta ordine tornano i nodi decisionali di **branch**, dove seguiamo uno specifico arco solo se la condizione che lo identifica si verifica. Per gli altri casi d'uso vedi le slide, lui non li ha spiegati.

Ci fermiamo qui dal momento che queste attività devono essere svolte da oggetti creati da classi che dobbiamo identificare, si passa quindi alla modellazione statica. Faremo uso stavolta di UML 2, dove una **modifica sostanziale** (oltre al fatto che il costrutto componente non rappresenta più la singola unità fisica ma vero e proprio elemento di progettazione) è stata effettuata sul class diagram mediante l'introduzione di **classi strutturate** (si specificano le classi in modo gerarchico!).

Partiamo da un class diagram chiamato context class diagram, dove sfruttando l'approccio gerarchico permesso dalle classi strutturate definiamo l'intera applicazione con una singola classe strutturata che viene annotata con lo stereotipo <<software system>>.

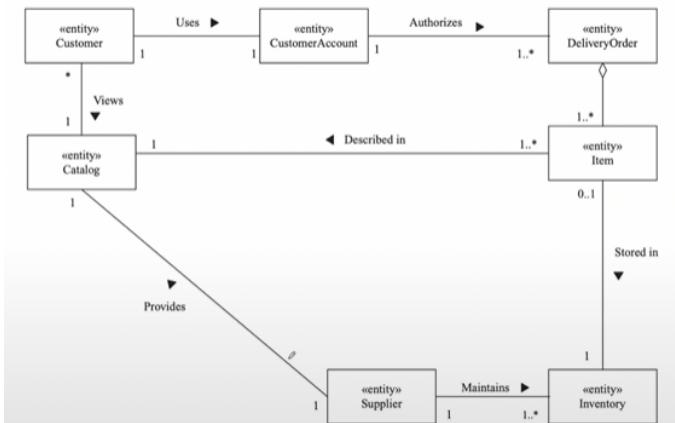


Si sfrutta quindi la classe *Online Shopping System* per rappresentare l'intero sistema. Questo approccio gerarchico come già accennato l'altra volta ricorda molto i **DFD Data Flow diagram**, dove si partiva da una rappresentazione del software dal

punto di vista del flusso dei dati usando un solo processo, che poi man mano veniva descritto in modo più dettagliato. (approccio non OO ma procedurale)

Facciamo lo stesso in questo caso: partendo dal context class diagram vediamo sempre più nel dettaglio il blocco Online Shopping System per capire cosa contiene. Per farlo, in modo similare a quanto fatto nel primo modulo, analizziamo i requisiti per determinare anzitutto le classi entity, le associazioni e cardinalità.

Entity Class Diagram



Customer, CustomerAccount, Supplier, Catalog, Inventory, DeliveryOrder.

Queste entità sono ricavate sia secondo il metodo dei sostantivi (approccio noun-phrase) dai requisiti sia dallo user case diagram, per cui sappiamo che i due attori supplier e customer devono essere inseriti come classi.

Per le associazioni osserviamo come si espliciti anche un verso (es. il cliente usa il suo account e vede il catalogo, il catalogo è visto dal cliente...), ciò serve a facilitare la lettura del diagramma. Poi vabbé le cardinalità, fissato un cliente questo ha associato un solo catalogo, fissato un catalogo ha associati 1...N (*) clienti etc...

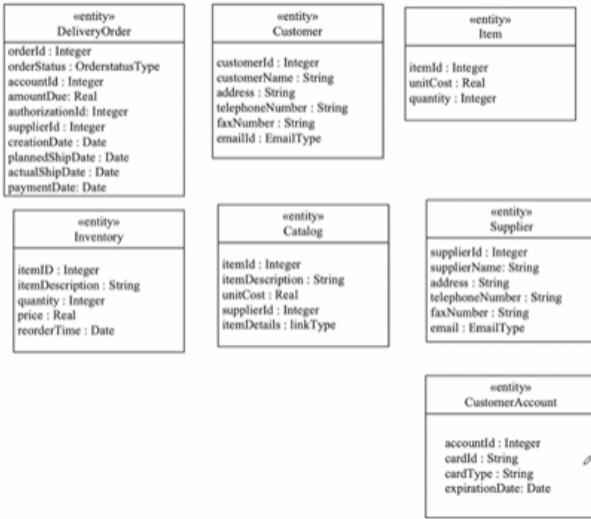
Poi associazione 1 a 1 tra cliente e account cliente, ricordiamo che le associazioni 1:1 potrebbero essere anche fuse in una singola classe (una singola istanza è legata a una singola istanza in entrambe le direzioni).

Poi a destra in alto DeliveryOrder vediamo che viene usato il simbolo del rombo vuoto, esso ricordiamo rappresentare una relazione di contenimento aggregation (composition rombo pieno per cui existence dependency se cancello il contenitore cancello anche il contenuto, aggregation rombo vuoto per cui semantica di riferimento, l'oggetto contenuto continua a esistere perché se cancello contenitore cancello solo il riferimento).

Perché in questo caso deliveryorder contenitore di item come aggregation?

Perché ovviamente i prodotti ordinati se l'ordine è cancellato potrebbero essere ordinati da altri clienti.

Per ogni classe entity definiamo ora i relativi attributi, esattamente in linea con quanto fatto nella prima parte del corso.

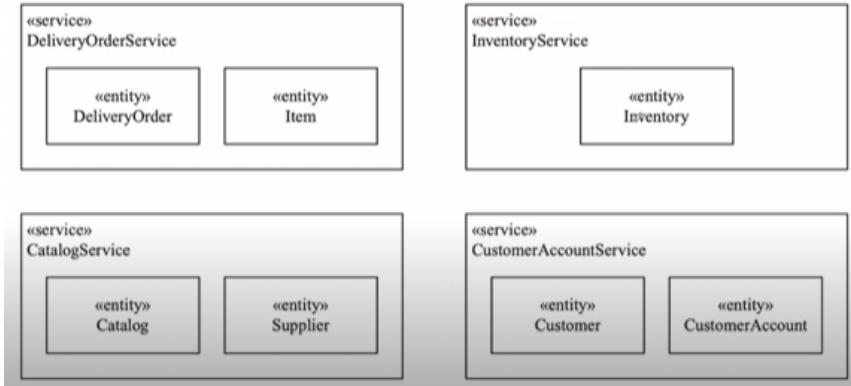


Da qui in poi iniziamo a differenza della prima parte del corso a introdurre il concetto di **classe strutturata** grazie all'uso di UML 2.

Essendo un'applicazione service oriented, le entity classes sono integrate nell'architettura utilizzando classi note come service classes.

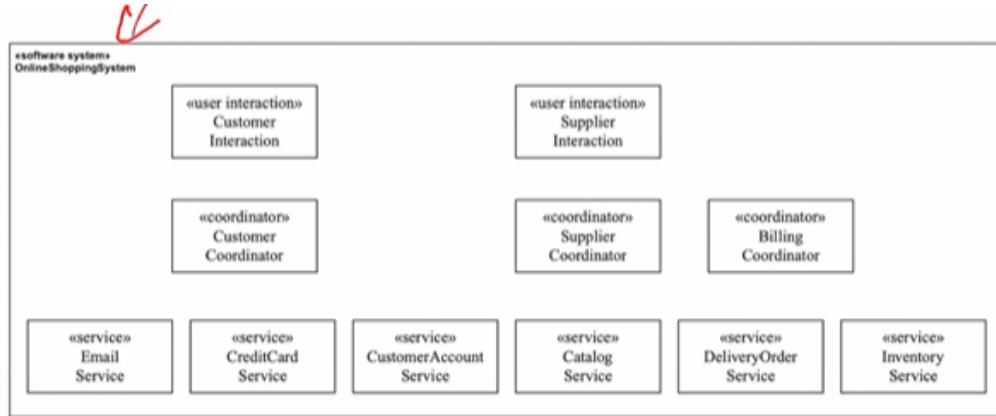
Si delineano 4 service classes: Catalog Service, Customer Account Service, Delivery Order Service e Inventory Service. Esse permettono l'accesso alle entity classes.

Vediamo come disporle:



Tuttavia le classi service non sono sufficienti per la nostra applicazione, in quanto esse permettono l'accesso alle classi entity ma non avremo solo entity, ma anche boundary e control. Si parla quindi di altre classi: *User Interaction Classes* (per **Boundary**) necessarie per l'interazione con gli utenti esterni (in particolare customer interaction e supplier interaction, quindi una classe per attore) e *Coordinator Classes* per gestire il coordinamento per l'invocazione delle service classes (in questo caso customer coordinator per gestire le richieste del cliente e supplier coordinator per le azioni del supplier, infine un billing coordinator per coordinare l'attività di pagamento per l'ordine da parte del cliente).

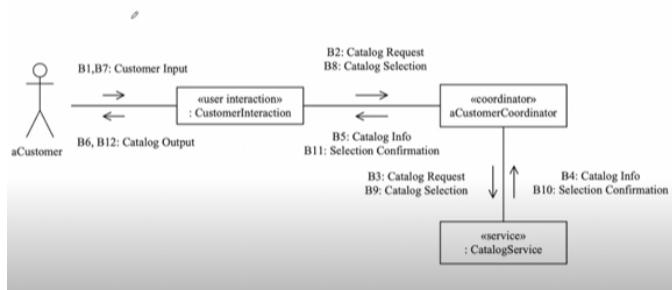
A questo punto la classe strutturata identificata inizialmente come Online Shopping System, che rappresentava l'intero software, è descritta in modo molto più dettagliato come segue:



Ricapitolando Classi Service per accedere alle classi entity ma anche a ulteriori servizi, infatti vediamo come nel diagramma ci siano a destra le 4 classi service identificate prima mentre a sinistra *due aggiuntive: una per il pagamento via carta di credito e una per inviare i messaggi via mail* (ciò suggerisce come si utilizzino due servizi esterni per integrare ciò, i nostri 4 dovremo implementarli mentre gli altri due li useremo come servizi esterni blackbox tipo gmail e paypal, senza che li implementi ex novo).

A questo punto torniamo al modello dei dati, bisogna infatti specificare come le funzionalità descritte nel diagramma delle attività vengono realizzate in termini di interazione tra oggetti. Quindi da un lato diagramma di casi d'uso con diagramma delle attività, poi **Static Modeling** per identificare le classi e a questo punto **Dynamic Modeling**, dove invece che usare i sequence diagram come fatto nella prima parte del corso utilizzeremo i **communication diagrams** (ex collaboration diagrams).

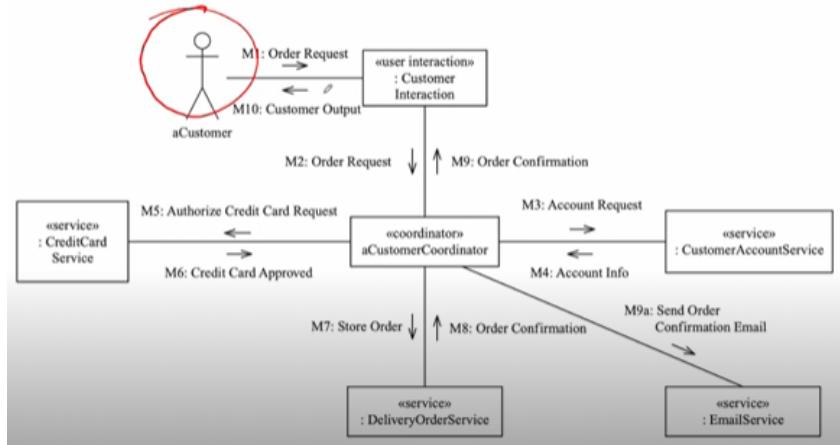
Communication diagram for the *Browse Catalog* UC



Non vi è in questo caso la lifeline tipica dei sequence diagram, non è rappresentata in modo esplicito la sequenza temporale di come sono scambiati i messaggi ma si può

tranquillamente ricostruire visto che ad ogni messaggio scambiato è associato un sequence number.

L'attore customer invia l'input a un oggetto di una classe di tipo user interaction, essa interagisce con il coordinator il quale a sua volta interagisce con la classe di servizio (abbiamo quindi in qualche modo ricostruito l'approccio BCE)

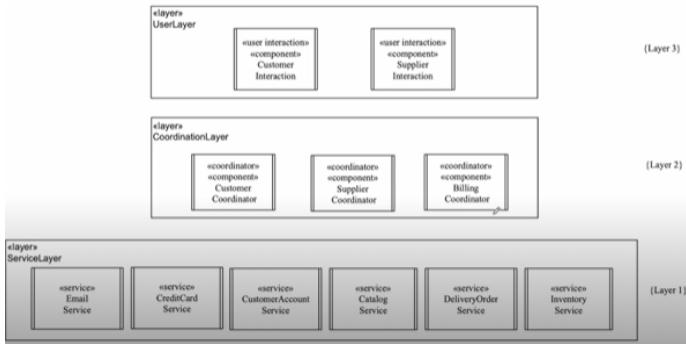


Le cose si complicano con il caso d'uso Make Order Request. Si noti comunque come l'approccio BCE sia sempre rispettato: l'attore interagisce con una user interaction class che interagisce con il coordinator che si occupa di prendere i dati dalle service classes. **Ricordiamoci come CreditCard Service ed Email Service sono classi che non verranno ulteriormente sviluppate in quanto servizi sviluppati da altri, sarà definita solo l'interfaccia necessaria per poterli utilizzare.**

Si ha nelle slide un communication diagram per ogni caso d'uso, in ognuno si rispetta la stratificazione dettata da BCE (boundary può interagire solo con boundary o control, control solo con control o entity, entity solo con entity o control).

A questo punto passiamo alla parte di **progettazione** (Design Modeling), dove andremo più nel dettaglio relativamente agli elementi che dobbiamo progettare ed implementare.

Dobbiamo anzitutto identificare i vari componenti, da strutturare in modo opportuno, per progettare l'applicazione service oriented. **Un componente sappiamo essere un elemento basato su un principio di netta separazione tra interfaccia e implementazione** -> per ogni componente da definire l'interfaccia e per quelli che effettivamente implementeremo da definire dettagli di progettazione e implementazione (ci fermeremo ovviamente alla parte di progettazione).



Partiamo dalla architettura stratificata, dove vediamo i tre strati rappresentati in modo esplicito (User, Coordination e Service, corrispondono al BCE).

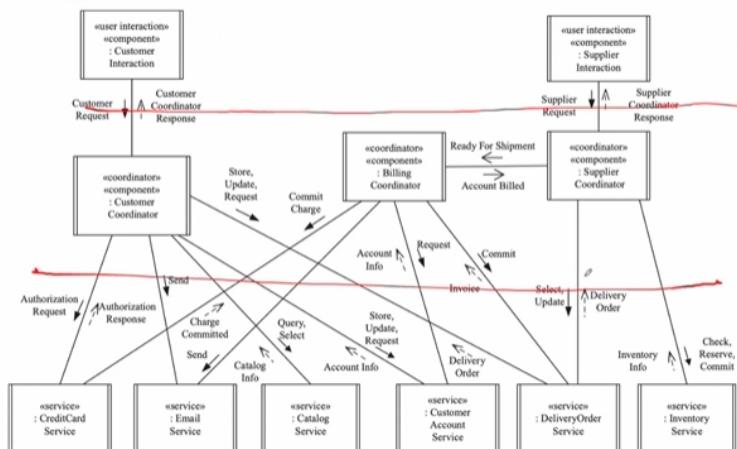
Quando si fa uso di architetture Service Oriented, sappiamo che facciamo uso di alcuni Pattern per garantire la comunicazione tra oggetti.

I pattern che definiremo a livello architetturale in questo caso saranno in particolare, il **Broker Forwarding Pattern** (per cui ogni richiesta che va dal cliente al fornitore passerà attraverso il broker per cui per ogni interazione quattro messaggi), poi due meccanismi di comunicazione, uno sincrono e uno asincrono (nel primo chi fa la richiesta si blocca in attesa della risposta), **Service Discovery** per scoprire quali servizi forniscono funzionalità di mail o addebitamento per carte di credito e **Two Phase Commit** per quel che riguarda eventuali transazioni.

Per questo caso di studio si farà maggiormente uso della comunicazione sincrona, con svantaggio il fatto che il client sarà sospeso nell'attesa di risposta dal servizio. Un'alternativa è utilizzare la comunicazione asincrona con callback da parte del servizio quando egli è pronto a inviargli la risposta, e questa scelta sarà utilizzata per la progettazione sia del Supplier Coordinato che del Billing Coordinator, mentre per il Customer Coordinator le interazioni saranno gestite con comunicazione sincrona.

Viene rappresentato il communication diagram, definito “concurrent” in quanto i rettangoli hanno barre laterali che indicano come ogni istanza eseguita dal communication diagram è eseguita indipendentemente dalle altre.

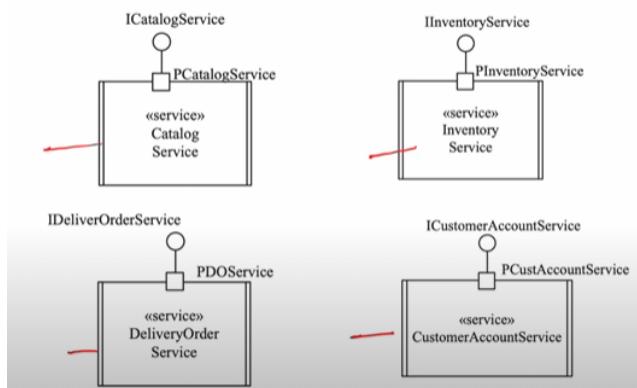
Concurrent Communication Diagram



Sono rappresentati tutti i messaggi scambiati tra le tre stratificazioni user, coordinator e service.

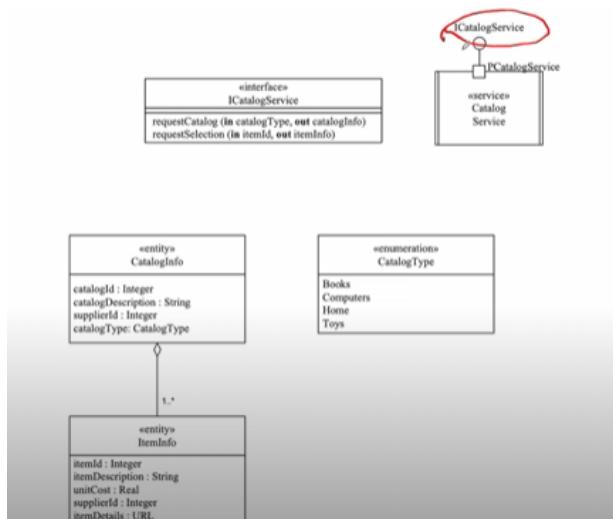
In UML ricordiamo che **call sincroni** e **signal messaggi asincroni**. *In UML 2 questa differenza è resa esplicita usando frecce piene per chiamate sincrone, normali per asincrone*. Si nota come anticipato che nel nostro esempio quasi tutte sincrone salvo la comunicazione con il Billing coordinator che è asincrona.

Progettare una componente significa definirne interfaccia e implementazione. Questo quindi lo facciamo per ognuno dei servizi che dobbiamo implementare:



UML permette di rappresentare l'interfaccia fornita da ciascun componente usando la notazione PALLINO, collegato alla classe attraverso una porta (coerentemente alla rappresentazione concettuale di servizio visto come black box alla quale accedo attraverso la porta) (IcatalogService interfaccia servizio catalogo, Pcatalogservice porta servizio catalogo).

A questo punto dobbiamo definire le interfacce del servizio.



L'interfaccia di Catalog Service è costituita da due operazioni: richiedicatalogo e richiediselezione (richiedere la selezione di un certo insieme di elementi).

Ogni operazione è fornita con la lista di parametri formali, e per ognuno di essi si esplicita se sia di input o output.

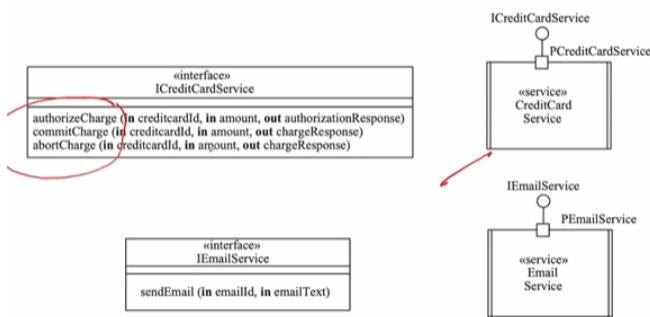
Dal punto di vista progettuale CatalogService permette di accedere alle classi CatalogInfo e ItemInfo, si rappresentano quindi anche gli attributi delle classi e la relativa associazione di aggregazione.

Anche la classe enumeration CatalogType per indicare i tipi di catalogo messi a disposizione dal fornitore.

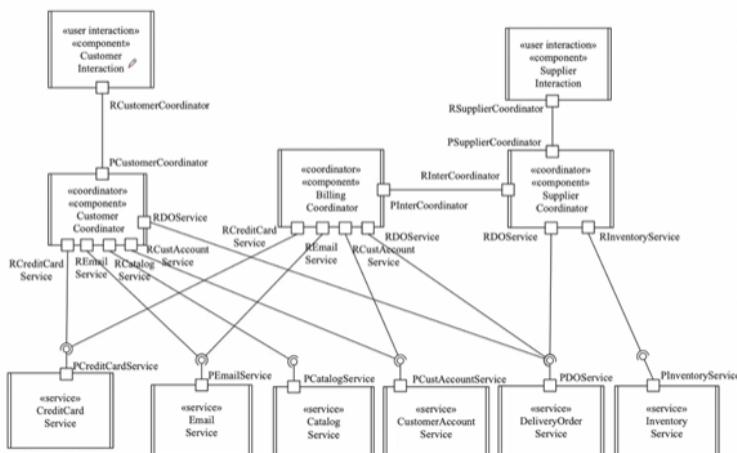
Si fa la stessa operazione (non solo interfaccia, ma anche classe di origine e classi con cui interagirà attraverso le operazioni, ossia classi contenute!) anche per customeraccountservice, il deliveryorderservice e l'inventoryservice.

Le altre due classi per la posta elettronica e per i pagamenti non li progettiamo noi, quindi ci limitiamo a descriverne l'interfaccia

Service Interface for Credit Card and Email services



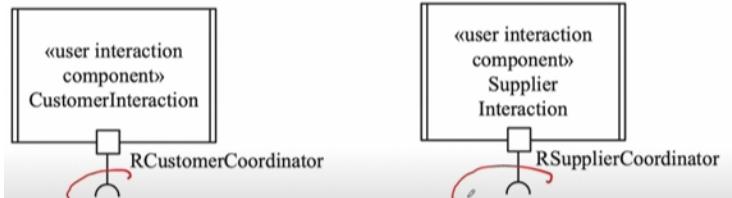
La nostra architettura è quindi costituita da una serie di componenti, ognuna con la propria interfaccia. Le componenti sono collegate dalle porte e si effettuano richieste definite secondo un **prefisso R (required)**, ossia il componente richiede una certa funzionalità o un **prefisso P (provided)**. Chiaro come **R per boundary** e **P per controllo** come risposta in quanto per l'interfaccia si richiede il servizio al coordinatore.



In basso notazione Lollipop, si utilizza per rendere ancora più esplicito il fornitore e il richiedente. Se sopra la palla del lollipop richiedente, se stecco del lollipop fornitore (vedi tra Pinventoryservice e RinventoryService, c'è una roba che

sembra un lollipop).

Ci si aspetta quindi che tutte le componenti User Interaction non vi sia alcuna funzionalità offerta (ha solo bisogno della componente coordinator)



Le “forchette” indicano che sti servizi necessitano solo funzionalità e non ne offrono.

Sul lato coordinator invece entrambe le cose: sopra si forniscono funzionalità alle user interaction, sotto si richiedono funzionalità alle componenti di servizio.

Lato service invece solo servizi messi a disposizione (pallino).

A questo punto bisognerà andare nel dettaglio per ogni componente di modo da descrivere come essa deve essere realizzata limitatamente a solo quelli che dobbiamo realizzare.

Questo è importante perché in ottica di riusabilità dell'applicazione, infatti una volta implementati i nostri servizi come il catalog service o il delivery order service possiamo metterli a disposizione su rete e diventare noi stessi fornitori di servizi.

Quindi per applicazioni service oriented vale lo stesso discorso fatto per le component based: se trovo le componenti già fatte che fanno al caso mio allora le prendo, altrimenti le implemento e poi le metto a disposizione per eventuali clienti futuri.

Con ciò abbiamo terminato la parte legata al caso di studio, dalla prossima volta aspetti sempre di progettazione ma più di dettaglio.

Lez 28 (03/04) (pure sta parte merda rivedila con chat)

Si inizia a vedere l'uso dei **Design Patterns**.

A partire da questo argomento dovrà estendersi il progetto, applicando almeno due Design Patterns al caso di studio.

Quando abbiamo parlato dei *principi di progettazione del software* e in particolare della **riusabilità**, abbiamo detto che i design pattern rappresentano uno degli artefatti che si riutilizzano in fase di progettazione.

Infatti i design pattern sono stati introdotti proprio per incrementare la riusabilità in fase di progettazione software, riducendo così tempi, costi e incrementando anche l'affidabilità!

Si vuole quindi definire il modello dei dati (diagramma delle classi) non solo correttamente, ma anche in modo quanto possibile riusabile!

Ciò è chiaramente complesso, soprattutto per software di dimensioni significative.

Sappiamo che in fase di analisi dei requisiti si rappresentano come classi entità che hanno una vera e propria controparte nel mondo reale, in fase di progetto emergono anche classi che invece non hanno alcuna controparte nel mondo reale (vedi classi boundary o di controllo), e le astrazioni che emergono in fase di progetto in questo senso sono fondamentali per rendere il progetto riusabile.

Ciò che fanno i Design Pattern è aiutare ad identificare queste astrazioni insieme alle classi che possono rappresentare, e l'idea di base è individuare soluzioni a problemi che sono ricorrenti in fase di progettazione, che siano il più possibile soluzioni generiche affinché si possa garantire la riusabilità.

Quindi un Design Pattern è uno strumento che incapsula una soluzione progettuale ad un problema ricorrente nella progettazione software.

Tra le **principali caratteristiche dei Design Patterns**: *rappresentano soluzioni a problematiche ricorrenti che si incontrano durante le fasi di sviluppo software* (ci concentreremo sulla fase di progettazione), *si sfrutta l'esperienza di progettisti anziani OOD* (OOD perché nel nostro caso object oriented) *che hanno trovato soluzioni facilmente riusabili, evitano al progettista di pensare a soluzioni che già sono state trovate da altri, permettono la scrittura di codice maggiormente comprensibile in quanto linguaggio comune* (soluzioni già codificate da altri e quindi conosciute), *semplificano la manutenzione.*

Ovviamente l'uso di design pattern non risolve tutti i problemi, ma cerca di alleviare la **risoluzione a problemi ricorrenti tramite soluzioni già identificate**.

Sono stati introdotti vari tipi di Design Pattern nel tempo, noi ne vedremo solo alcuni citati nel libro della GANG OF FOUR "Design Patterns" (vabbé).

Per capire quale Design Pattern fa al nostro caso, è necessario classificarli, e per farlo si sfruttano vari criteri.

Il primo criterio è **l'obiettivo (Purpose)** del design pattern, distinguiamo tre classi:

- **Creazionali**: pattern utilizzati per facilitare operazioni di creazione oggetti
- **Strutturali**: utilizzati, sfruttando le caratteristiche OO come ereditarietà e polimorfismo, aiutano a definire la struttura del sistema *in termini di composizione classi e oggetti* in modo consapevole
- **Comportamentali**: si concentrano sul modellare il comportamento del sistema, **nel nostro caso OO** quindi sul come far interagire gli oggetti che lo popolano

Il secondo criterio riguarda il **raggio di azione (scope)**, ossia su cosa si applicano i pattern. Due possibilità:

- **Classi**: i pattern lavorano a livello di classi e quindi definiscono le relazioni tra classi e sottoclassi (*basate sul concetto di ereditarietà e quindi relazioni statiche*, ossia definite a tempo di compilazione)
- **Oggetti**: pattern che definiscono le relazioni tra oggetti, *relazioni che quindi possono cambiare anche durante l'esecuzione e che per questo sono dinamiche.*

Nel libro della Gang of Four si trova un vero e proprio catalogo di pattern, per un totale di **23**. Ciò che faremo è *illustrare almeno un pattern per ogni possibile combinazione di purpose e scope*.

		Scopo		
		Creazionale	Strutturale	Comportamentale
Raggr. d'azione	Classi	Factory Method	Adapter (class)	Interpreter Template Method
		Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of responsibility Iterator Mediator Memento Observer State Strategy Visitor

In figura quelli in grassetto sono i 9 pattern che effettivamente illustreremo.

Oltre ad essere classificato, ciascun pattern viene descritto usando un formato standard, che presenta un totale di 10 campi:

- **Nome e Classificazione** (nome tipicamente significativo per descrivere l'essenza del pattern, classificazione scope e purpose)
- **Motivazione**: descrive il perché il pattern è stato introdotto e quindi il problema che deve risolvere
- **Applicabilità**: descrive le situazioni in cui il pattern può essere applicato
- **Struttura**: descrive lo schema di soluzione, ossia graficamente la configurazione GENERICA di elementi che risolvono il problema (relazioni, responsabilità, collaborazioni...).

Starà al progettista che farà uso del design pattern partire dallo schema generico per adattarlo al progetto specifico.

- **Partecipanti**: classi e oggetti che fanno parte del pattern con relative responsabilità
- **Conseguenze**: risultati che si ottengono applicando il pattern
- **Implementazioni**: tecniche e suggerimenti utili all'implementazione del pattern
- **Codice di esempio**: frammenti di codice che illustrano come implementare il pattern in un certo linguaggio di programmazione (java o c++ tipicamente)
- **Usi conosciuti**: esempi di applicazione in sistemi reali
- **Pattern Correlati**: tipicamente non si applica un solo pattern ma spesso è possibile combinarne l'uso

Ogni pattern che descriveremo quindi sarà presentato con questo formato.

Prima di passare ai pattern ricordiamo un concetto utile, i **Framework**.

Un Framework non rappresenta una semplice libreria, ma un insieme di classi che cooperano al fine di fornire lo scheletro di un'applicazione riusabile in uno specifico dominio applicativo (molto utilizzati per la realizzazione di editor visuali, particolari domini tipo finanziario etc...), il framework è lo scheletro di un'applicazione che viene personalizzata da uno sviluppatore.

I **framework** consentono di definire la struttura statica e lo scopo generale di un sistema software, **facilitando il riuso sia a livello di design sia, in parte, a livello di codice**. Il riuso avviene soprattutto nella fase di progettazione, poiché il **framework fornisce lo scheletro dell'interazione tra gli oggetti**.

Diversamente da una libreria, che viene utilizzata richiamando esplicitamente le sue classi o funzioni quando necessario, un framework impone una struttura predefinita in cui il codice sviluppato dall'utente viene integrato e invocato dal framework stesso. Questo meccanismo è spesso descritto come "inversione del controllo", poiché non è l'utente a chiamare il framework, ma è il framework a chiamare il codice dell'utente nei punti previsti.

I framework sono basati, come sappiamo, sul concetto di **Classe Astratta**. Una classe astratta è una classe che fa parte del framework e ha dei metodi che però non sono implementati, dovranno esserlo dallo sviluppatore nel caso specifico -> **il framework "guida" all'implementazione**.

Abbiamo ripassato il concetto di framework in quanto i Design Pattern aiutano molto a costruire e definire Framework.

Chiaramente Pattern diversi da Framework per tanti motivi, primo tra tutti il fatto che i **Pattern sono infatti elementi più piccoli a livello architettonico rispetto ai framework** (un framework contiene più pattern, ma il contrario non avviene mai) ed inoltre **i pattern sono meno specializzati dei framework** (framework per specifici domini applicativi, design pattern generici affinché siano applicabili in tutti i domini).

Iniziamo a vedere i pattern partendo da **scope Creazionale**.

Abstract Factory: **pattern creazionale basato su oggetti**.

- **Scopo**: fornire un'interfaccia per la creazione di famiglie di oggetti, ossia oggetti correlati o dipendenti tra loro, senza specificare le loro classi concrete. **Questo consente di creare oggetti che devono essere utilizzati insieme in modo coerente**.

- **Motivazione**: si consideri il caso in cui si desidera sviluppare un'applicazione con un'interfaccia utente in grado di supportare diversi *look & feel* (ad esempio una versione chiara e una scura). Gli elementi dell'interfaccia — come finestre, pulsanti e menu — mantengono la stessa funzionalità, ma devono poter essere visualizzati in stili differenti.

Per rendere l'applicazione portabile tra diversi stili visivi, è importante evitare il **cabling** (ovvero il vincolo rigido) degli oggetti grafici nel codice. Se ogni componente dell'interfaccia viene istanziato direttamente con una classe concreta (es. DarkButton, LightWindow), modificare l'aspetto richiederà cambiamenti estesi nel codice. Il pattern **Abstract Factory** permette invece di creare questi oggetti tramite un'interfaccia comune, rendendo facile il passaggio da uno stile all'altro.

Vediamo un esempio concreto. Si considera uno strumento per realizzare UI dove si vuole garantire compatibilità con due look&feel.

Il primo prevede l'utilizzo di finestre create con la classe Window1 con relativa scrollbar ScrollBar1, e il secondo lo stesso ma per Window2 e ScrollBar2.

Chiaramente l'applicazione deve realizzare una UI che rispetti le relazioni tra oggetti e sia portabile da un Look&Feel a un altro -> **il client non deve istanziare direttamente gli oggetti ma deve far riferimento a qualcun altro che istanzi gli oggetti in base a un determinato Look&Feel e inoltre bisogna rispettare le relazioni tra oggetti, evitando che il client accoppi (sbagliando) ad es. Window1 e Scrollbar2.**

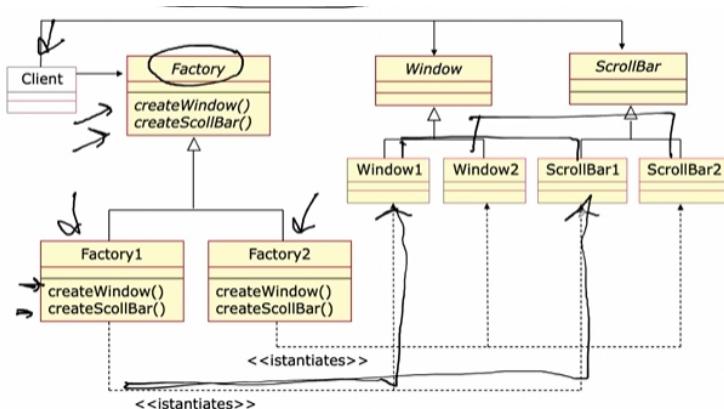
Per riuscire a far ciò si utilizza quindi l'Abstract Factory. Partiamo dall'esempio per poi vedere la struttura generale.

L'applicazione client ha a che fare con le 4 classi Window1, ... ScrollBar 2.

Affinché le classi siano associate in modo corretto si evita che sia il client a istanziare oggetti a partire da esse. **Si utilizza quindi un'abstract factory responsabile della creazione di windows e scrollbar, a cui il client farà riferimento!**

La fabbrica astratta è implementata attraverso la Factory1 e 2 (necessarie per esprimere le 2 combinazioni possibili in questo caso).

Quindi se il client volesse usare Look&Feel1, si riferisce alla Factory1 che saprà come istanziare correttamente gli oggetti.



Se non si utilizzasse l'abstract factory, il client dovrebbe avere la responsabilità di accoppiare correttamente gli oggetti, mentre se usa abstract factory creerà un'oggetto da quella classe astratta a cui sarà quindi demandata la responsabilità.

- Senza utilizzare l'Abstract Factory l'applicazione client deve esplicitamente istanziare gli oggetti.
- Il rispetto delle relazioni è cablato nel codice e deve essere noto al client.

```
Window w = new Window1();
...
ScrollBar s = new ScrollBar1();
```

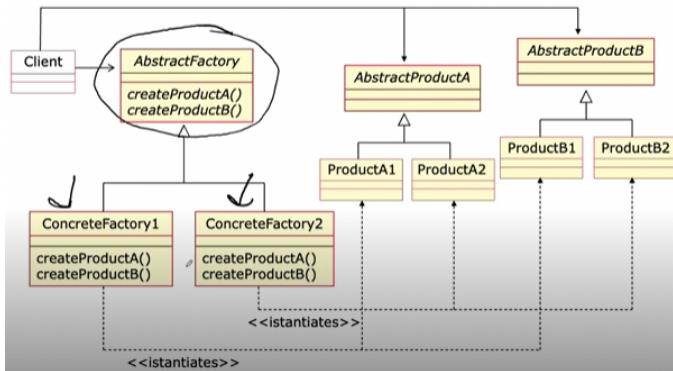
- Con l'Abstract Factory la responsabilità è demandata alla Factory.

```
Factory f = new Factory1();
Window w = f.createWindow();
...
ScrollBar s = f.createScrollBar();
```

La struttura generale dell'abstract factory presenta la classe astratta che fa riferimento a una serie di n prodotti, per ognuno dei quali esiste una specifica implementazione.

L'abstract factory deve essere implementata da una fabbrica concreta di oggetti, che indirizzerà i prodotti corretti accoppiandoli correttamente (chiaramente in generale n prodotti e altrettante fabbriche concrete). - **Struttura:**

Abstract Factory: struttura



Tornando alle caratteristiche standard dell'abstract factory:

- **Applicabilità:** si fa riferimento a software dove il client non deve sapere come vengono creati gli oggetti, in particolare è utile quando il sistema è configurabile con famiglie di prodotti diverse ed il client non è legato a una specifica famiglia (ossia può sceglierne una come un'altra).

- **Partecipanti:** elementi partecipanti nella struttura -> AbstractFactory, ConcreteFactory, AbstractProduct, ConcreteProduct, Applicazione Client

- **Conseguenze:** grazie all'abstract factory si isolano le classi astratte dalle concrete, le famiglie di prodotti possono essere rapidamente cambiate perché la factory completa compare in un unico punto del codice, si tratta di un pattern ovviamente applicato in fase di progettazione quindi se volessi aggiungere ulteriori famiglie dovrei ricompilare il codice (l'insieme di prodotti gestiti è legato all'interfaccia della factory).

Il prossimo pattern ha sempre scopo **creazionale**, ma stavolta lo **scope non è su oggetti ma su classi**: **Factory Method**:

- **Scopo:** il Factory Method definisce un'interfaccia per la creazione di un oggetto, delegando alle sottoclassi la decisione su quale classe concreta istanziare. In questo modo, la creazione dell'oggetto avviene in modo dinamico, a tempo di esecuzione.

A differenza dell'Abstract Factory, dove l'ereditarietà è usata in modo più statico (aggiungere una nuova factory richiede la modifica e la ricompilazione del codice), il Factory Method consente una maggiore **flessibilità e estensibilità**, permettendo al codice principale di rimanere invariato mentre le sottoclassi decidono il tipo specifico di oggetto da creare.

- **Motivazione:** il Factory Method è ampiamente utilizzato nei framework, dove le classi astratte definiscono la struttura e le relazioni tra gli elementi del dominio, ma delegano alle sottoclassi la responsabilità della creazione degli oggetti concreti. Questo consente ai framework di definire il flusso generale dell'applicazione, lasciando flessibilità all'utente nell'estendere e personalizzare il comportamento senza modificare il codice base.

Si consideri l'esempio di un framework per la gestione di documenti di tipo diverso. Le due astrazioni chiave del framework sono *Application* (applicazione principale) e *Document* (classe astratta che rappresenta il concetto di documento, ma non il tipo specifico).

Sono gli sviluppatori che usano il framework a dover definire delle sottoclassi per ottenere implementazioni adatte al loro caso specifico -> l'associazione tra applicazione e documento specifico da creare è nota solo a tempo di esecuzione -> application sa quando creare un documento (dopo una specifica istruzione) ma non sa esattamente quale deve creare (questo è delegato alle sottoclassi definite dagli utilizzatori del framework).

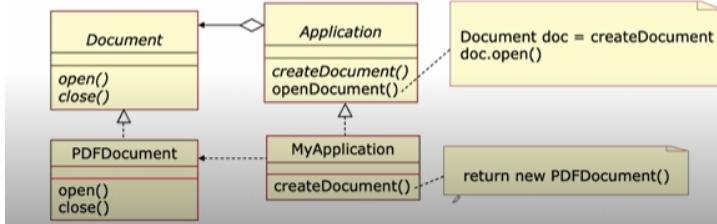
Grazie al Factory Method, il framework delega la decisione sulla **classe concreta da istanziare** alla sottoclasse di Application implementata dallo sviluppatore. È così che l'associazione viene risolta **dinamicamente, a tempo di esecuzione**.

In figura: il Document è classe astratta in questo caso implementata da un particolare tipo di documento (PDF), l'Application è invece responsabile della creazione del documento. In essa è presente un metodo astratto createDocument() e uno concreto openDocument() implementato usando la nota in alto a dx (invoca createDocument e poi lo apre). Il metodo createDocument() essendo astratto è implementato dalla sottoclasse specifica che sa esattamente quale tipo di documento deve esser creato (in questo caso PDF).

Perché il Design Pattern si chiama Factory Method? Perché si utilizza piuttosto che una classe astratta un Metodo “Fabbrica”, metodo astratto che dovrà essere implementato dalle specifiche sottoclassi poi responsabili della creazione dello

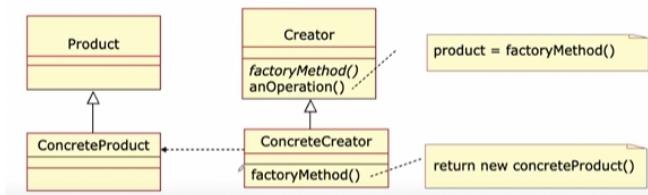
specifico tipo di documento nel nostro caso.

- Il pattern Factory incapsula la conoscenza della specifica classe da creare al di fuori del framework



- Struttura:

Factory Method: struttura



In generale l'operazione `factoryMethod()` (che fa parte della classe astratta `Creator`) delegherà la creazione dell'oggetto del prodotto specifico a una sottoclassa realizzata dagli utilizzatori del framework (qui si vede chiaramente il riuso “inverso” dei framework per cui non chiamo un metodo che mi è reso disponibile, ma mi viene fornito uno scheletro che poi io completo per i miei casi specifici).

Poi a sinistra il prodotto concreto, che sarà istanziato dall’implementazione concreta del `factoryMethod()`.

- **Applicabilità:** si applica quando una classe non è in grado di sapere in anticipo le classi di oggetti che deve creare (tipico dei framework dove si sa quali operazioni si vogliono fare ma non su quale tipo di oggetti farle), quando la classe vuole che siano sottoclassi specifica a scegliere gli oggetti da creare -> le classi delegano la responsabilità di creazione

- **Partecipanti:** `Product`, `ConcreteProduct`, `Creator` e `ConcreteCreator`

- **Conseguenze:** elimina la necessità di riferirsi a classi dipendenti dall’applicazione all’interno del codice -> **il codice creator è altamente riusabile perché poi applicabile su varie sottoclassi specifiche che saranno definite dagli utilizzatori del framework.**

Design pattern quindi strumenti generici che risolvono problemi ricorrenti nella progettazione software (facilmente riusabili).

Ora vediamo il **Pattern Adapter**, pattern di **obiettivo Strutturale** unico che può essere **applicato sia con scope Object che con scope Class**.

- **Scopo**: *serve a convertire l'interfaccia di una classe esistente incompatibile con quanto necessario al client in una versione compatibile.*

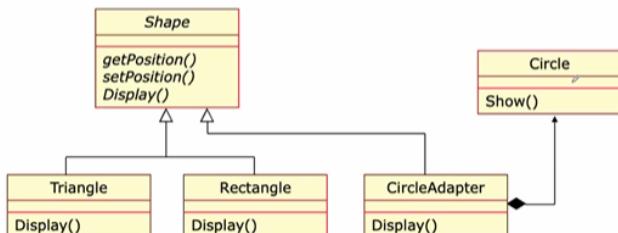
- **Motivazione**: si consideri un editor che consente di disegnare e comporre oggetti grafici. L'astrazione chiave è un singolo oggetto grafico. *Si suppone di voler integrare un nuovo oggetto grafico che già abbiamo a disposizione ma che non ha interfaccia compatibile con l'editor.*

Si utilizzerà quindi adapter per utilizzare il nuovo oggetto grafico senza dover reimplementare tutte le singole funzionalità.

Astrazione chiave nell'esempio: Forma (oggetto grafico) sul quale posso invocare metodi per sapere ad esempio la posizione, settarla o rappresentarlo graficamente. La classe astratta Forma è poi implementata da Triangle e Rectangle, che in particolare implementano il metodo display. *Si suppone poi di avere a disposizione una classe Circonferenza che però è stata implementata con interfaccia differente (non ho Display ma Show).* Ora o implemento una nuova classe Circle con metodo Display reimplementato a nuovo o adatto il metodo Show() che già ho a disposizione.

Due soluzioni: Adapter con scope a Oggetti

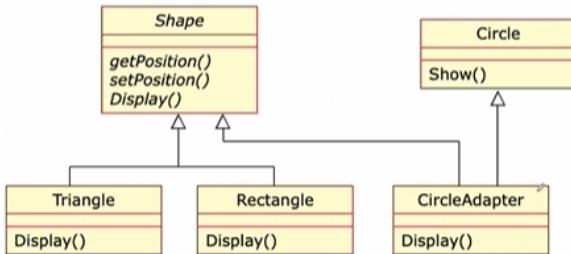
♦ Soluzione 1: Object Adapter



Si realizza una classe chiamata non Circle ma CircleAdapter che crea l'oggetto Circle (riusando la classe Circle), e quando viene invocato il metodo Display() nell'adattatore sarà invocato il relativo metodo Show() per Circle.

Si utilizza la Composition UML (rombo pieno) in quanto appunto l'oggetto Circle è creato all'interno dell'oggetto CircleAdapter -> è encapsulato.

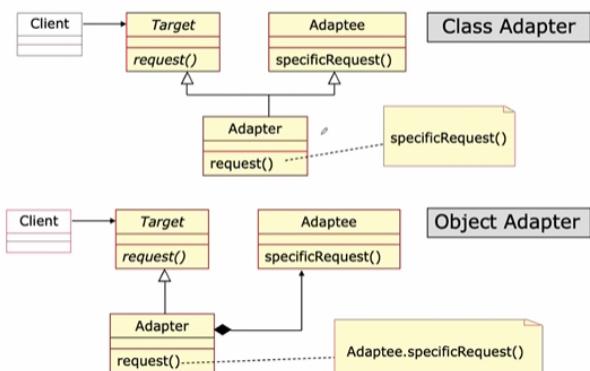
La seconda soluzione è **l'Adapter con Class Scope**, dove invece di usare la composition si usa l'ereditarietà



CircleAdapter eredita sia da Shape che da Circle (*ereditarietà multipla, non si potrebbe utilizzare direttamente per Java, potrei però risolvere il problema vedendo Shape come interfaccia quindi uso implements e Circle come superclasse quindi extends*), il metodo Display() ridefinisce il metodo Show() facente parte della classe Circle.

- Struttura:

Adapter: struttura



Si ha l'interfaccia Target a cui dobbiamo adattarci, la classe Adapter adattatore e Adaptee da adattare. Con Class Adapter si fa uso dell'ereditarietà (si chiama il metodo della superclasse), con Object Adapter composizione (si crea l'oggetto dall'Adapter su cui viene invocato il metodo specificRequest()).

- Applicabilità: *si usa quando siamo interessati a riusare una classe esistente ma con interfaccia incompatibile a quella desiderata*

- Partecipanti: *Client, Target, Adapter e Adaptee*

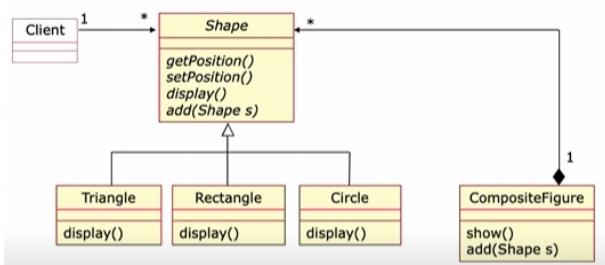
- Conseguenze: **è necessario prendere in considerazione l'effort necessario all'adattamento** (*dobbiamo comunque creare una classe adapter, bisogna quindi capire se conviene farlo piuttosto che creare direttamente una nuova classe adatta per l'interfaccia*).

Vediamo ora tra il pattern **Composite**, che ha **purpose strutturale e scope basato su oggetti**.

- **Scopo**: **comporre oggetti in strutture gerarchiche che consentano di trattare i singoli elementi e la composizione in modo uniforme** (allo stesso modo)

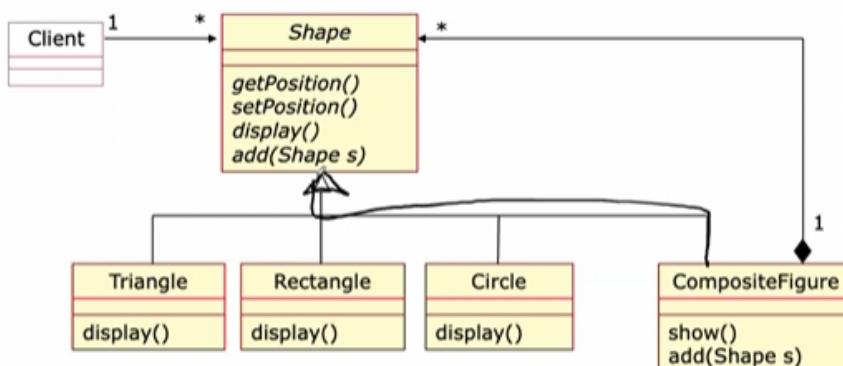
- **Motivazione**: *Le applicazioni grafiche consentono di trattare in modo uniforme sia le forme geometriche di base (linee, cerchi etc...) sia oggetti complessi che si creano a partire da questi elementi semplici. Molti editor grafici hanno ad esempio la funzione raggruppa, dove data una serie di elementi l'editor lo considererà come un unico elemento atomico.*

Si considera come esempio una app grafica in grado di gestire gli oggetti elementari Triangle, Circle e Rectangle. *Si ha come requisito che l'applicazione permetta il raggruppamento dinamico di oggetti elementari in oggetti composti. I due tipi di oggetti devono essere trattati allo stesso modo (in modo uniforme).*

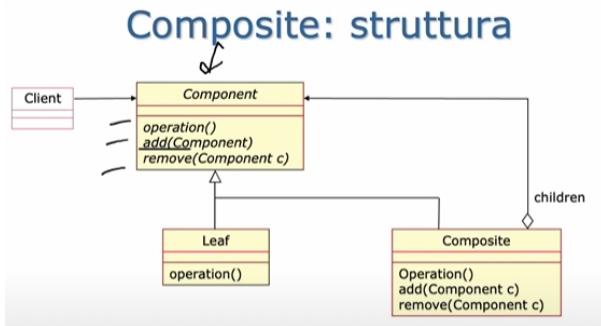


Il client interagisce con la classe Forma. *Rispetto a prima abbiamo anche il metodo add che permette di aggiungere a un elemento altri elementi.* Oltre agli oggetti elementari, abbiamo un'altra classe FiguraComposita che può essere costituita da 1 o più forme (infatti cardinalità *), fissata una FiguraComposita questa può essere costituita da molteplici forme). Costrutto di Composizione in quanto la FiguraComposita è costituita da 1 o più Figure, e se muore l'oggetto da CompositeFigure muoiono anche gli oggetti che fanno parte di lui (li creo direttamente al suo interno, non per riferimento)

L'utilizzo di questa soluzione però non permette di creare altre FigureComposite a partire da una FiguraComposita (è collezione di oggetti elementari ma non è essa stessa figura geometrica) -> *non sono soddisfatti tutti i requisiti -> è necessario introdurre ereditarietà anche per CompositeFigure*



- Struttura:



Il Client interagisce con il componente generico avente una certa interfaccia, classi elementari (Leaf) che costituiscono direttamente Component (possono essere quindi più di uno) e Composite ricordandomi di usare non solo il costrutto di aggregazione o composizione ma anche ereditarietà.

Se si usa composition se si cancella la figura composita si cancellano anche gli elementi contenuti all'interno, se usiamo aggregation posso cancellare il raggruppamento senza cancellare gli oggetti raggruppati.

- Applicabilità: si usa quando si vogliono rappresentare le gerarchie di oggetti in modo che oggetti semplici e compositi siano trattati allo stesso modo

- Partecipanti: *Component e Composite, Leaf, Client*

- Conseguenze: *i client sono semplificati perché gli oggetti semplici e compositi sono trattati allo stesso modo* (interagisce solo con l'interfaccia Component senza dover conoscere l'interfaccia dell'oggetto composito), *l'aggiunta di nuovi oggetti Leaf o Composite è relativamente semplice sfruttando il codice dell'applicazione già esistente*. Tuttavia può rendere il sistema troppo generico e poco flessibile, con questa soluzione infatti non posso creare ad esempio un oggetto composito composto solo ed esclusivamente da triangoli e rettangoli, potrei sempre aggiungerci cerchi (per farlo dovrei svincolarmi dall'interfaccia Shape mettendo relazioni di composizione solo con Triangle e Rectangle)

NB Factory scope a oggetti perché la creazione di oggetti è delegata a sottoclassi mentre abstract factory a oggetti perché la creazione degli oggetti è delegata agli oggetti.

Lez 29 (08/04)

(breve riassunto) Negli anni come detto sono stati proposti vari Design Patterns, noi stiamo facendo riferimento solo ad alcuni di quelli standard, ossia presenti nel libro della gang of four.

*Abbiamo visto come il Factory Method sia design pattern per cui si introduce un metodo utilizzato dal client per la creazione di oggetti il cui tipo non è noto al tempo di compilazione, per cui la sua creazione è **delegata** alla sottoclasse che implementa la superclasse con il metodo factory, mentre l'Abstract Factory svincola il client dalla responsabilità di creare famiglie di oggetti -> il client interagisce con un oggetto factory che si occuperà di creare gli oggetti a lui necessari e quindi sarà quell'oggetto ad assumersi la responsabilità della creazione esplicita di questi oggetti.*

Poi Adapter che ha come obiettivo l'adattare, e quindi riusare, classi esistenti che però non sono conformi all'interfaccia che ci serve. Senza implementare ex novo la classe esistente si cercano modi per adattarla, il pattern si applica sia con scope class che object usando gli strumenti dell'OO ereditarietà e composizione (per scope object si incapsula l'oggetto da riusare usando il meccanismo di composizione, per scope class si usa l'ereditarietà per creare un oggetto conforme alla nostra interfaccia e userà il metodo della superclasse (che è la classe da adattare) per renderla conforme all'interfaccia.)

Infine pattern Composite molto utile se si devono gestire gerarchie di oggetti trattandoli però tutti allo stesso modo.

Si passa ora al successivo design Pattern: il **Decorator**. *Si tratta di un pattern con purpose strutturale e scope basato su oggetti.*

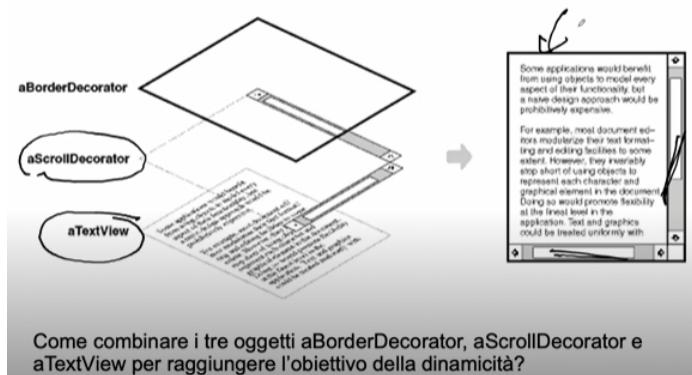
- Scopo: *si vogliono “decorare” gli oggetti. Una decorazione rappresenta l'aggiungere dinamicamente (runtime) funzionalità (responsabilità) ad un oggetto.*
Il **subclassing** (per cui a partire da una classe generale ne definisco di più specifiche che ereditano tutto e in più aggiungono funzionalità) **rappresenta un'alternativa**

statica (compile time, non runtime) a ciò, il cui scope è a livello di classe e non di singolo oggetto!

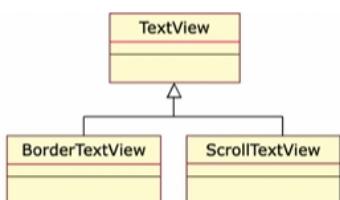
- **Motivazione:** si fa riferimento allo scenario classico di *realizzazione di un'interfaccia utente di tipo grafico*. L'idea è di avere oggetti di base, come una finestra, a cui si vogliono aggiungere funzionalità come ad es. testo scorrevole o particolare bordo

Immaginiamo di avere tre oggetti. Si parte dall'oggetto base, una finestra di testo (TextView) al quale vogliamo aggiungere delle responsabilità (“decorazioni”) come ad esempio la barra di scorrimento ScrollDecorator e un bordo BorderDecorator. **Ma come raggruppare questi tre oggetti al fine di raggiungere l'obiettivo di dinamicità** (cioè a tempo di esecuzione poter decorare la finestra con gli altri due oggetti)?

Decorator: esempio



L'approccio classico che scartiamo è quello basato su subclassing, dal momento che come detto prima è approccio statico.



Estendendo la classe finestra con classi specifiche che ne aggiungono bordo e scorrimento sto adottando un approccio statico poiché lo definisco a tempo di compilazione (devo mette mano sul codice), e inoltre devo preoccuparmi di creare una sottoclasse per ogni possibile funzionalità che potrei esser interessato ad aggiungere, ed inoltre se volessi creare una finestra che abbia entrambe le funzionalità dovrei creare una specifica sottoclasse BorderAndScrollTextView (un puttanaio insomma) -> troppe combinazioni! 🤯

A questo punto, torniamo al concetto di **composizione**.

L'idea di decorare un oggetto, ovvero **arricchirlo con funzionalità aggiuntive**, può

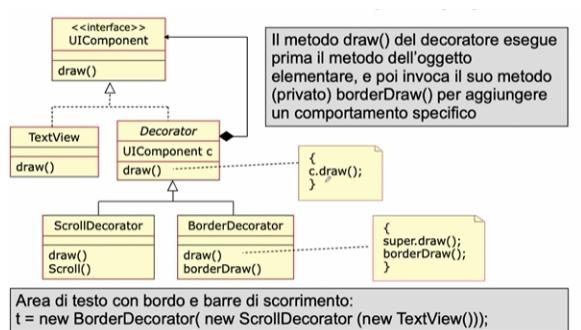
essere realizzata in modo molto più flessibile utilizzando un approccio compositivo: **un oggetto “base” viene incapsulato all’interno di un altro oggetto**, detto **Decorator**, che ne estende dinamicamente il comportamento.

Il **Decorator** si comporta come un involucro: **implementa la stessa interfaccia** dell’oggetto che intende decorare, così da poter essere utilizzato in modo intercambiabile. Internamente, il decorator **inoltra (delegando) le chiamate** all’oggetto incapsulato, ma può anche eseguire **comportamenti aggiuntivi prima o dopo la delega**.

Per esempio, se l’oggetto base rappresenta una finestra grafica, un decorator potrebbe aggiungere dinamicamente una **barra di scorrimento** o un **bordo**, senza modificare il codice della finestra stessa.

Si ha un’interfaccia che è il nostro componente generico di User Interface, essa ha il metodo `draw()` per visualizzare il componente a livello grafico.

`TextView` e `Decorator` implementano il metodo generico di interfaccia (fanno `draw()`) e `Decorator` in più incapsula `UIComponent` -> incapsula qualunque oggetto che implementa `UIComponent`, quindi anche `TextView`.



Tuttavia `Decorator` rappresenta ancora una classe astratta -> da essa estendo `ScrollDecorator` e `BorderDecorator`, in particolare ognuna di esse aggiunge il metodo necessario per estendere le funzionalità base dell’oggetto elementare (`TextView`). **Quando si esegue il `draw()` del decoratore, prima si esegue il `draw()` dell’oggetto elementare (`TextView`) e poi si invoca il metodo per aggiungere il comportamento specifico (es. `borderDraw()`) per aggiungere una funzionalità specifica. Da `borderDraw()` si evoca il `draw` della superclasse (`Decorator`) che a sua volta invoca il `draw` del componente che incapsula (`TextView`) (NB la incapsula perché incapsula l’interfaccia -> incapsula tutte le classi concrete che la implementano) -> viene prima disegnata la finestra di testo e poi si aggiunge il bordo.)**

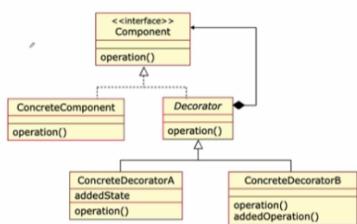
Questo meccanismo è molto flessibile, in quanto se voglio creare un oggetto che ha sia bordo che barra di scorrimento faccio `t = new BorderDecorator(new ScrollDecorator(new TextView()))` (`TextView` oggetto base è incapsulato in

ScrollDecorator, e l'oggetto così ottenuto viene ulteriormente incapsulato in BorderDecorator aggiungendo quindi l'ulteriore responsabilità del bordo).

In questo modo posso aggiungere tutte le funzionalità che voglio senza implementare in modo statico tutte le combinazioni di funzionalità come sottoclassi.

- Struttura:

Decorator: struttura



L'interfaccia Component descrive un componente generico, poi classe astratta Decorator implementata da tutte le possibili decorazioni (A, B, ...). Ogni volta che creo un oggetto ConcreteDecorator dovrò specificare come parametro del costruttore l'oggetto che dovrà essere decorato (ConcreteComponent).

- Applicabilità: *si applica se necessario aggiungere responsabilità a oggetti in modo trasparente e dinamico, quindi quando in particolare il subclassing (statico) non è adatto*

- Partecipanti: *Component e ConcreteComponent, Decorator e ConcreteDecorators*

- Conseguenze: *maggior flessibilità rispetto all'approccio statico, evita quindi di definire strutture gerarchiche complesse.*

Alcune note aggiuntive: il Decorator è molto usato in Java, in particolare nella definizione degli stream Input/Output:

```
BufferedInputStream bin = new BufferedInputStream( new FileInputStream(  
    "test.dat"));
```

(in questo caso il bufferedInputStream decora un fileInputStream di base).

Il decorator sembra simile al pattern Composite al livello strutturale, ma il composite lo abbiamo usato per gestire strutture gerarchiche (creando raggruppamenti di oggetti) mentre decorator serve ad aggiungere funzionalità in modo dinamico -> scopi diversi.

Decorator sembra simile anche al pattern Adapter, ma quest'ultimo si limita ad un adattamento di un'interfaccia senza aggiungere alcuna funzionalità.

Tra le principali limitazioni del Decorator il fatto che le decorazioni possono essere applicate più e più volte (es. metto il bordo alla finestra con scorrimento), ciò che però non è vietato è applicare la stessa funzionalità più e più volte (es. metto il bordo sul bordo sul bordo sul bordo...). Ciò non ha senso dal punto di vista della funzionalità ma è consentita dal pattern.

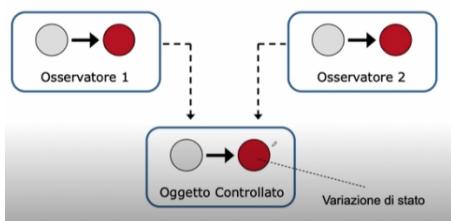
Ora vediamo il **Pattern Observer**, uno dei più usati. È **pattern comportamentale basato su oggetti**.

- **Scopo:** *definire una dipendenza uno a molti tra oggetti, mantenendo basso il grado di coupling.*

Il nome Observer deriva dall'idea di realizzare uno strumento che permetta di gestire facilmente l'accesso ad un oggetto che cambia stato da parte di un insieme di oggetti osservatori interessati a questo cambiamento di stato, in modo che possano aggiornarsi automaticamente.

- **Motivazione:** lo scenario classico è quello con GUI ossia applicazioni con interfaccia grafica secondo il paradigma Model-View-Controller (== BCE), per cui ad esempio se si vuole mostrare via interfaccia il cambiamento dei dati se oggetti Model (entity) cambiano, allora gli oggetti che implementano la View (boundary) devono aggiornarsi.

Observer: l'idea di fondo



Es. software che mostra i risultati di una partita di calcio, gli oggetti legati al risultato possono cambiare dinamicamente stato in tempi anche molto ristretti, si vuole fare in modo che quando lo stato cambia gli osservatori (come le semplici interfacce grafiche che mostrano all'utente questi risultati) siano modificati aggiornando il valore.

Vediamo una prima (naive) soluzione: si potrebbero utilizzare nell'oggetto osservato attributi pubblici o metodi pubblici (getters) che leggono il valore di un attributo protetto -> periodicamente gli osservatori controllano se vi sono stati aggiornamenti.

Non si tratta tuttavia di una buona soluzione, in quanto non è scalabile (se troppi osservatori l'oggetto osservato è sovraccaricato dalle richieste), gli osservatori dovrebbero continuamente interrogare l'oggetto osservato e variazioni rapide potrebbero non essere comunque rilevate da qualche osservatore (se le richieste sono fatte ad esempio con timestamp di 10 secondi allora per 9 secondi ci si potrebbe

perdere un risultato se quello avviene 1 secondo dopo la richiesta da parte dell'osservatore).

Lez 30 (10/04)

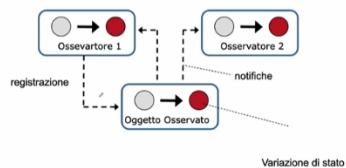
L'approccio corretto è quello per cui invece sia l'oggetto osservato a dover notificare gli osservatori in caso di cambiamento.

Il pattern **Observer** in particolare prevede che gli osservatori si registrino presso l'oggetto osservato, in questo modo sarà lui a notificare ogni cambiamento di stato agli osservatori.

Quando l'osservatore rileva la notifica, può interrogare l'oggetto osservato oppure svolgere operazioni indipendenti dallo specifico stato.

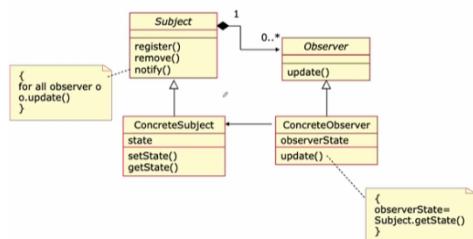
Observer: l'approccio corretto (2)

- Gli osservatori possono essere aggiunti a runtime



- Struttura:

Observer: struttura



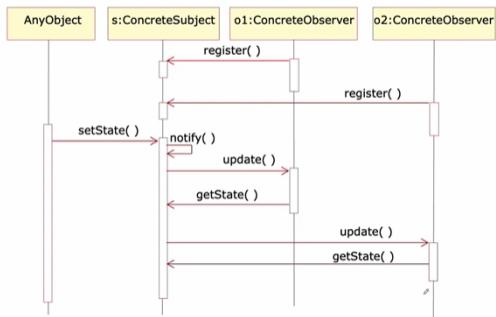
Subject è l'oggetto da osservare, l'observer è l'oggetto interessato ai cambiamenti di stato. Entrambe sono classi astratte che mettono a servizio i metodi register() e remove() (Subject) che utilizzano gli observer per mostrare o meno interesse verso un subject, notify() è invece un metodo usato dallo stesso Subject per notificare gli oggetti interessati di cambiamenti di stato. Il metodo update() è il metodo eseguito dal Subject verso tutti gli observer registrati dopo notify().

Dire che avviene un cambiamento di stato significa dire che il ConcreteSubject (ossia colui che effettivamente implementa la classe astratta) esegue setState(), e dopo averlo eseguito esegue notify() il cui corpo come vediamo in figura ci dice che per ogni observer registrato esegue o.update().

Dopodiché sarà l'observer in questione che nel metodo update() chiamerà il metodo getState() dal subject per recuperare il nuovo valore dello stato.

Il workflow appena descritto è chiaro guardando il seguente Sequence Diagram:

Observer: sequence diagram



s, o1, o2 rappresentano istanze. Ogni altro oggetto dell'applicazione chiama `setState()` sull'oggetto osservato facendogli quindi cambiare stato. Quindi il `concretesubject` invoca su se stesso il metodo `notify()` che poi invoca l'`update` su tutti gli oggetti che si erano registrati. Dopodiché l'osservatore, se interessato, effettuerà il `getState()` sull'oggetto osservato.

- **Applicabilità:** molto utile per mantenere basso il livello di coupling (la soluzione naïve vista prima invece prevedeva coupling molto alto), infatti riduciamo il numero di messaggi (solo in caso avvenga effettivamente un cambiamento di stato). Utile quindi per gestire le modifiche di oggetti conseguenti la variazione dello stato di un oggetto.

- **Partecipanti:** `Subject, Concretesubject, Observer, ConcreteObserver`

- **Conseguenze:** l'accoppiamento tra `Subject` e `Observer` è astratto, infatti il `Subject` conosce solo la lista degli osservatori. Inoltre la notifica è una comunicazione di tipo broadcast (il `subject` non si occupa di quanti sono gli `observer` registrati).

Bisogna porre attenzione dal momento che una qualsiasi modifica al `Subject` (`setState()`) comporta una serie di messaggi e modifiche a tutti gli osservatori.

Rappresenta uno dei Pattern più utilizzati.

Template Method è un **Pattern comportamentale basato su classi**.

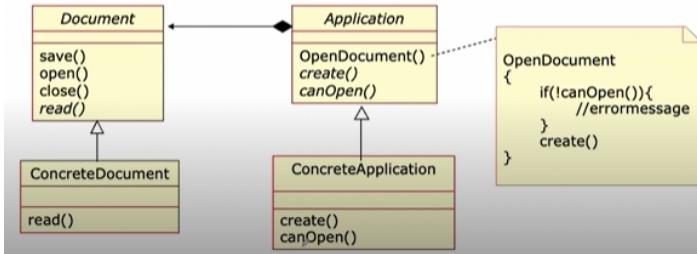
- **Scopo:** **Principalmente utilizzato nei Framework, permette di definire la struttura di un algoritmo interno ad un metodo delegando alcuni dei suoi passi alle sottoclassi.**

- **Motivazione:** Si consideri un **framework per la costruzione di applicazioni** che gestiscono **documenti di vario tipo** (es. testo, Word, PDF, ecc.).

In questo contesto, viene utilizzato il **pattern Template Method** per definire un

algoritmo generale, delegando alle sottoclassi la definizione dei dettagli specifici.

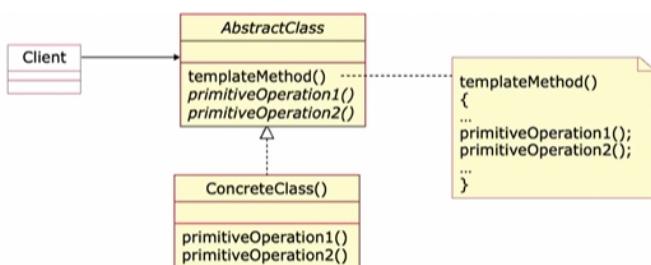
- ◆ Consideriamo quindi un semplice framework costituito da 2 classi: Application e Document.



Si hanno due classi astratte: **Document**, che definisce le operazioni comuni sui documenti: open(), save() e close(). Tra queste, il metodo read() è astratto, poiché la sua implementazione **varia in base al tipo di documento**: ad esempio, leggere un file Word richiede un algoritmo diverso da quello per leggere un file PDF. Saranno quindi le sottoclassi (es. WordDocument, PdfDocument, ecc.) a fornire l'implementazione concreta di read(). L'altra classe astratta è **Application**, che definisce il comportamento generale dell'applicazione per quanto riguarda la gestione dei documenti. Include metodi come create, canOpen e OpenDocument() dove openDocument() rappresenta il **Template Method**: definisce lo **schema dell'algoritmo** per aprire un documento, e al suo interno utilizza canOpen() per verificare se il documento può essere aperto.

Tuttavia, i metodi create () e canOpen () sono **metodi astratti**, e la loro implementazione è demandata alla sottoclasse concreta, ad esempio ConcreteApplication, in base al tipo specifico di documento da gestire.

- Struttura:



Il client invoca il metodo template che prevede una serie di operazioni, alcune delle quali però sono implementate da sottoclassi che concretizzano la classe astratta.

- **Applicabilità:** utilizzato per implementare la parte invariante di un algoritmo, lasciando alle sottoclassi la definizione degli step variabili. È utile quindi quando ci sono comportamenti comuni che possono esser inseriti nel template.

- **Partecipanti:** *AbstractClass, ConcreteClass, Client*.

- **Conseguenze:** permettono ovviamente riuso del codice, creano struttura di controllo invertito dove è la classe padre a chiamare le operazioni ridefinite dai figli e non viceversa (infatti normalmente l'uso che si fa dell'ereditarietà è che estendendo

le superclassi voglio riutilizzare quanto fatto da loro per fare di più, invece in questo caso non si segue proprio questa strategia).

I metodi richiamati dalla superclasse sono detti metodi **gancio** (hook), in quanto definiti nella parte variabile dell'algoritmo e legati quindi alla sua implementazione concreta.

Ci accorgiamo che TemplateMethod e FactoryMethod sono simili tra loro nell'approccio utilizzato, quali sono le differenze?

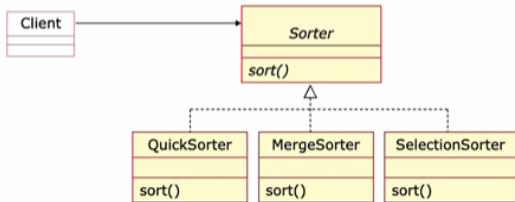
Infatti il FactoryMethod è un Pattern di tipo creazionale che si basa anch'esso sul delegare la responsabilità di creazione di un oggetto alle sottoclassi.

La differenza principale sta nello scopo: il Factory Method è metodo astratto che deve creare e restituire l'istanza di classe concreta, al fine di "deresponsabilizzare" il client dalla scelta del tipo specifico. Il Template Method d'altra parte è un metodo che invoca metodi astratti al fine di generalizzare un algoritmo.

Lo **Strategy Method** è infine un **pattern comportamentale basato su oggetti**.

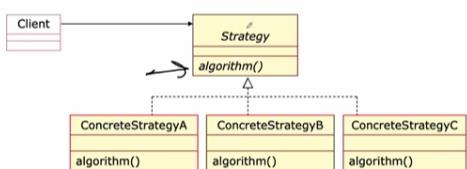
- **Scopo**: permette di definire famiglie di algoritmi in modo da renderli intercambiabili indipendentemente dal client che li usa.
- **Motivazione**: si consideri ad esempio la famiglia degli algoritmi di ordinamento (es. QuickSort, HeapSort, AmatoSort, etc..). Si vuole costruire un'applicazione che li supporti tutti e che sia facilmente estendibile se se ne volessero introdurre di nuovi, e che inoltre permetta una scelta rapida del miglior algoritmo da usare.

Gli algoritmi di ordinamento devono essere indipendenti dal vettore di dati su cui operano, e dal resto dell'implementazione dell'applicazione



Si implementa un'interfaccia Sorter poi implementata da un insieme di sottoclassi che realizza quegli algoritmi

Strategy: struttura



L'idea di fondo è disaccoppiare quindi il client dall'implementazione degli algoritmi e dal definire una logica di scelta sull'algoritmo da utilizzare in base al caso.

- **Applicabilità:** il pattern fornisce un modo per avere un'interfaccia comune tra algoritmi diversi di una stessa famiglia da utilizzare da parte del client.

- **Partecipanti:** *Strategy, ConcreteStrategy, Client*

- **Conseguenze:** il Pattern separa l'implementazione degli algoritmi dal contesto dell'applicazioni (se venissero implementati direttamente come sottoclassi della classe Client non sarebbe una buona scelta).

Inoltre in questo modo si eliminano i blocchi condizionali che sarebbero altrimenti necessari inserendo tutti i diversi comportamenti in un'unica classe.

Lo svantaggio principale sta nel fatto che i Client devono conoscere le diverse strategie.

Conclusa quindi la parte di Design Pattern.

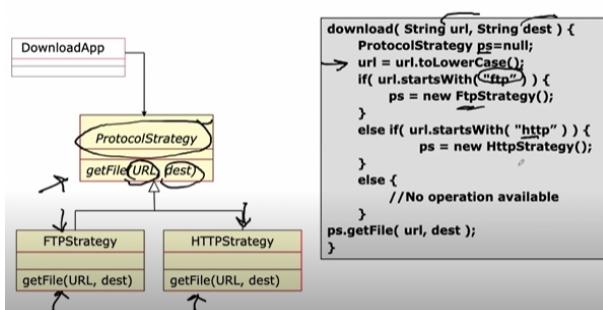
Vediamo ora delle applicazioni (esercizi) che fanno capire meglio il codice e il fatto che pattern diversi spesso sono utilizzati in modo combinato.

Esercizio 1: realizzare un'applicazione per gestire il download dai siti http e ftp. La selezione avviene in base all'utilizzo dell'url (ftp o http).

Ci viene suggerito di utilizzare factory method, per creare l'algoritmo di download e togliere la responsabilità dal client di scegliere quale algoritmo utilizzare in base al caso, e uno strategy per implementare il download secondo i due protocolli (si combinano quindi pattern creazionale con comportamentale!).

L'idea è scrivere un'applicazione che sia estendibile senza problemi (coerentemente con i concetti di manutenibilità e riusabilità).

L'interfaccia ProtocolStrategy fornisce un metodo getFile dove si specifica URL e destinazione del file, essendo limitata la possibilità di download a HTTP e FTP lo strategy è implementato da FTPStrategy e HTTPStrategy

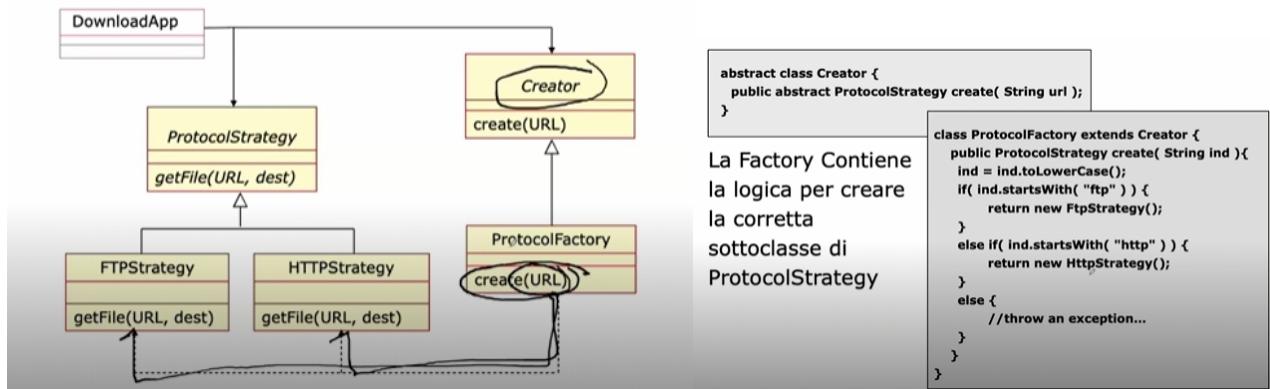


Il client è in questo caso colui che, come si vede nel codice a destra del metodo null, si occupa di "decidere" quale metodo usare per getFile (FTP o HTTP, lo capisce tramite if, else if).

Per aggiungere un eventuale nuovo protocollo si deve modificare la classe DownloadApp e il metodo download.

Inoltre DownloadApp in questo scenario deve conoscere dettagli implementativi come sapere esattamente i protocolli supportati.

Per evitare ciò si introduce una Factory.



Si utilizza secondo la logica vista per la Factory una classe astratta creator che fornisce un metodo astratto di creazione implementato dalla classe ProtocolFactory. Stavolta è la classe ProtocolFactory a contenere il metodo create contenente la logica prima associata al Client di dover scegliere che download usare in base al protocollo: infatti ora la Factory prende in input l'URL, se inizia per HTTP usa la strategia HTTP implementata da HTTPStrategy altrimenti FTPStrategy se inizia con FTP.

```

abstract class ProtocolStrategy {
    abstract void getFile(String ind, String dest);
}

class FtpStrategy extends ProtocolStrategy {
    public void getFile(String url, String dest) {
        //...method implementation
    }
}

class Download{
    ...
    download( String url, String dest ) {
        ProtocolFactory p = new ProtocolFactory();
        ProtocolStrategy ps = p.create(url);
        ps.getFile(url,dest);
    }
}
  
```

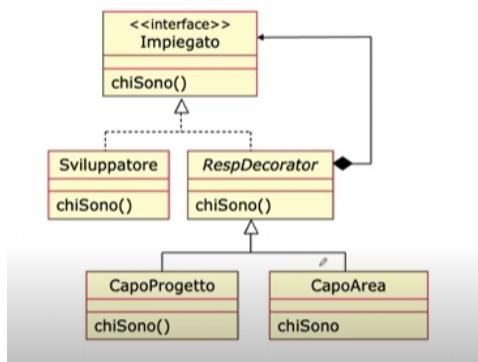
Ancora HTTPStrategy e FTPStrategy implementano il getFile in base al protocollo, ciò che veramente cambia è la classe Download, molto più compatta: si deresponsabilizza il client dal dover conoscere e “scegliere” quale metodo utilizzare grazie all’uso della Factory -> il client istanzia la protocolfactory, le chiede di creare l’oggetto per effettuare il download in base solo all’URL che prende in input e poi chiede di scaricare il file (ps.getFile) -> se volessi introdurre nuovi protocolli e quindi algoritmi di download mi basta agire sulle classi implementative di Strategy e ridefinire il metodo create in ProtocolFactory

Esercizio 2: Si vuole descrivere un modello a oggetti che rappresenti gli impiegati di un’azienda. Essi devono avere un metodo chiSono() per visualizzarne il nome e la responsabilità. Ogni impiegato può avere responsabilità aggiuntive come CapoArea

e/o CapoProgetto, ed una particolare categoria di impiegati che ci interessa sono gli Sviluppatori.

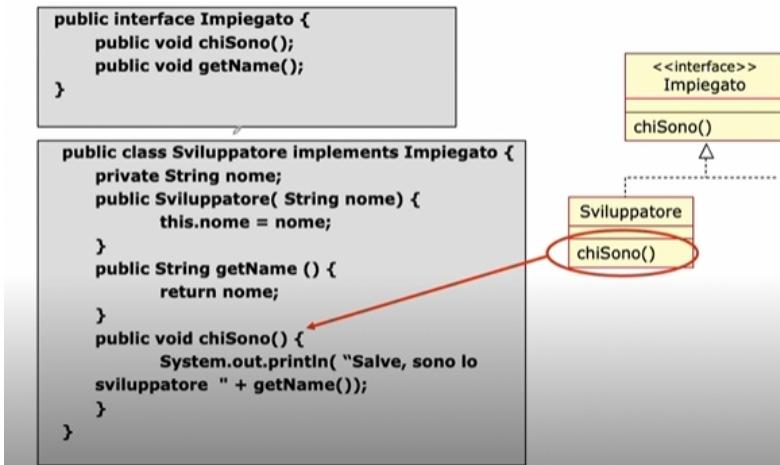
Viene suggerito di utilizzare il Decorator Pattern per assegnare dinamicamente le responsabilità (es. se uno sviluppatore diventa capoprogetto, devo poterlo rendere evidente in runtime) ed inoltre l'ereditarietà per contraddistinguere Sviluppatori da impiegati generici.

Dall'esame dei requisiti si può provare a disegnare una struttura con queste caratteristiche: deve sicuramente esistere una classe Impiegato che definisca un'interfaccia comune (metodo chiSono()), deve esistere una classe Sviluppatore che eredita da Impiegato e si devono definire via Decorator due classi ancora più specifiche: CapoArea e CapoProgetto.



(L'idea naïve poteva essere creare superclasse impiegato con sottoclassi sviluppatore, sviluppatore capoarea, sviluppatore capoprogetto, sviluppatore capoarea capoprogetto... non scalabile e tutti i problemi visti quando abbiamo parlato di Decorator).

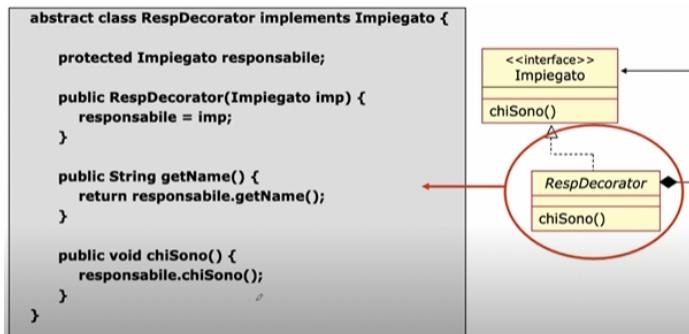
Si ha l'interfaccia principale di impiegato implementata dalla classe concreta sviluppatore e dalla classe astratta decorator, che a sua volta è implementata da capoprogetto e capoarea. Ricordiamo l'importanza dell'associazione di composizione tra **RespDecorator** e **Impiegato**: il decorator infatti funziona creando un oggetto decorato con l'oggetto da decorare; in fase di creazione di un oggetto **CapoProgetto**, lo "decoreremo" facendogli contenere un oggetto **Sviluppatore** (si ricorda anche il problema di decorator per cui se si usa in modo inappropriato si potrebbe creare uno sviluppatore con responsabilità due volte capoprogetto e così via...).



Vediamo come prima anche l'implementazione vera e propria.

La classe Sviluppatore implementa l'interfaccia Impiegato -> costruttore per cui alla creazione dello sviluppatore si deve inserire nome e metodo chiSono che oltre a nome restituisce la responsabilità che in questo caso è semplice sviluppatore.

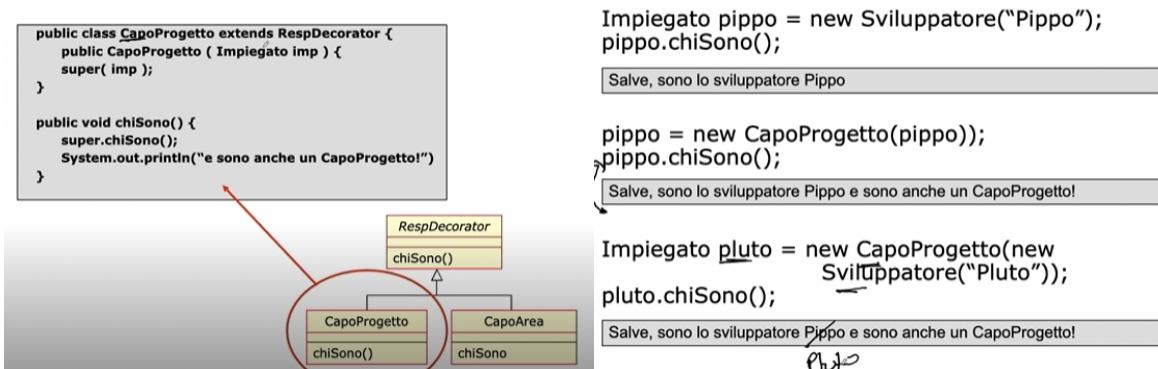
Vediamo ora come rendere lo sviluppatore anche capoprogetto e/o capoarea.



La classe astratta Decoratore implementa Impiegato, essendo chiaramente classe astratta non posso istanziare oggetti direttamente da questa classe.

L'attributo responsabile viene assegnato all'impiegato su cui viene applicata la decorazione, il metodo getName() ritorna l'utilizzo del metodo getName() sul responsabile in questione e chiSono() sarà utilizzato sull'implementazione concreta dell'impiegato.

Vediamo cosa succede per la decorazione concreta:



CapoProgetto estende RespDecorator (extends perché classe astratta), invoca il costruttore della superclasse e l'implementazione del suo metodo chiSono() consiste nell'invocare il metodo della superclasse aggiungendo la frase “e sono anche CapoProgetto” delineando la responsabilità aggiuntiva!

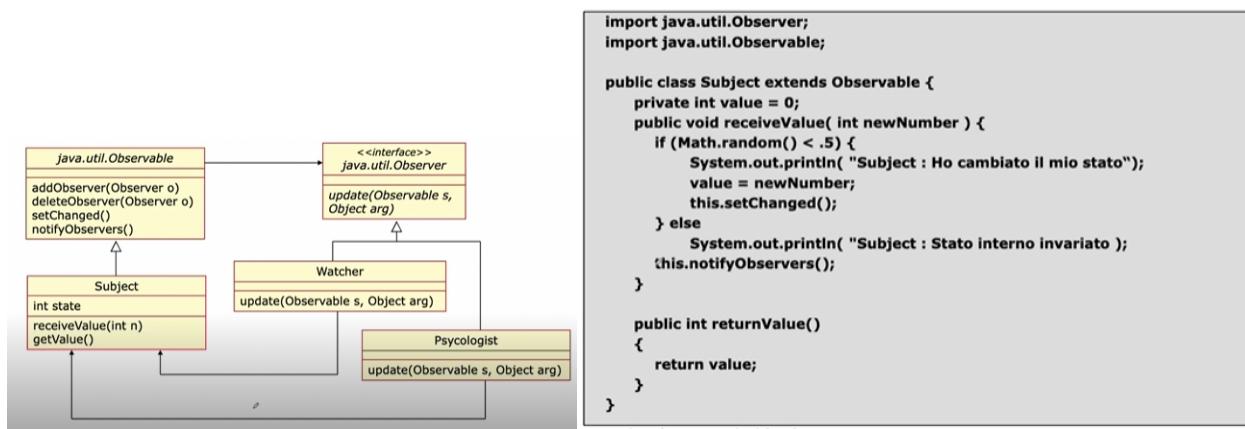
Quando si utilizza ciò l'utente crea l'oggetto Impiegato nominandolo come sviluppatore. Se poi lo si vuole render capoprogetto (decoro l'oggetto pippo come capoprogetto), per farlo creo l'oggetto CapoProgetto passando nel costruttore l'impiegato base.

Posso chiaramente continuare a decorare ad esempio l'oggetto pipp come CapoArea e risponderà adeguatamente (sono anche un Capoprogetto e sono anche un CapoArea)

Esercizio 3: si considera un oggetto Subject (da controllare) che riceve valori dall'esterno e in modo casuale cambia il suo stato interno accettando o meno il valore ricevuto.

Due oggetti Watcher e Psychologist devono monitorare il cambiamento di subject. Si suggerisce di utilizzare le interfacce Observer e Observable.

In Java sono presenti package che forniscono l'interfaccia Observer (con metodi standard per il pattern come update) e la classe astratta Observable che fornisce i metodi per registrare, notificare etc... (sarebbe il nostro subject)



Il subject implementa l'observable semplicemente usando un attributo che rappresenta lo stato (in questo caso un semplice int). ReceiveValue per ottenere eventuali valori e decidere se cambiare stato, getValue necessario per l'Observer affinché recuperi eventuali nuovi valori di fronte a una notifica.

In questo caso il modo per vedere se si cambia o meno lo stato di fronte ad un valore è che se un numero random generato tra 0 e 1 è $< 0.5 \rightarrow$ cambio lo stato con il numero che è stato passato altrimenti no. Dopodiché in ogni caso avviene la notifica verso gli observer. (NB invece di value nel codice dovrebbe essere state, invece di

public int returnValue() dovrebbe essere public int getValue() e poi return state, piccole inconsistenze).

```
import java.util.Observer;
import java.util.Observable;

public class Watcher implements Observer {
    private int changes = 0;

    public void update(Observable obs, Object arg) {
        System.out.println("Watcher: Lo stato del subject è: "
            + ((ObservedSubject) obs).returnValue() + ".");
        changes++;
    }

    public int observedChanges() {
        return changes;
    }
}
```

Si implementa per l'Observer il metodo update dove si ritorna lo stato del subject, si usa l'attributo changes per contare il numero cambiamenti dell'oggetto osservato.

```
public class Esempio {
    public static void main (String[] args) {
        ObservedSubject s = new ObservedSubject();
        Watcher w = new Watcher();
        Psychologist p = new Psychologist();
        s.addObserver(w);
        s.addObserver(p);

        for(int i=1;i<=10;i++){
            System.out.println("Main : Invio il numero " + i);
            s.receiveValue(i);
        }
        System.out.println("Il subject ha cambiato stato: " + s.observedChanges());
    }
}
```

```
Main : Invio il numero 1
Subject : Stato interno invariato
Main : Invio il numero 2
Subject : Ho cambiato il mio stato
Watcher: Lo stato del subject è: 2.
...
Il subject ha cambiato stato 4 volte.
```

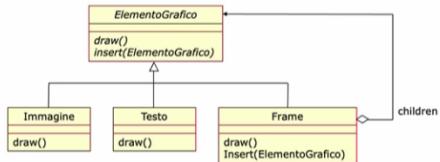
Qui l'esempio di come deve essere utilizzato il tutto lato client: si istanzia il subject, si istanziano i due osservatori watcher e psycho, poi si simulano dieci chiamate (via for) il metodo receiveValue ed ogni volta che si fa l'invocazione si stampa lo stato. Poi alla fine si utilizza l'observedChanges() per capire quante volte il subject ha cambiato lo stato.

Lez 31 (15/04)

Esercizio 4: si vuole realizzare un Editor in grado di gestire elementi grafici costituiti di elementi elementari come frame, testi e immagini. L'editor deve supportare due diversi algoritmi di formattazione e inoltre degli elementi grafici complessi come barre di scorrimento o bordi grafici.

Si suggerisce l'utilizzo di pattern Composite (per gestire elementi grafici sia elementari che compositi), Strategy (per i due algoritmi di formattazione), Decorator (per decorare gli elementi grafici con elementi aggiuntivi come barre di scorrimento/bordi).

Rappresentazione degli elementi grafici: il pattern Composite

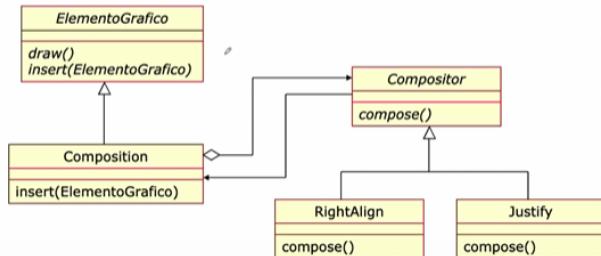


Nota: In Java le classi foglia (Immagine e Testo) devono necessariamente implementare il metodo insert() previsto dalla classe astratta. In questo caso una possibile soluzione è che queste generino una eccezione apposita

Il frame è componente grafico che raggruppa altri componenti e eventualmente anche altri frame -> usiamo Composite usando l'associazione di composizione/applicazione (nel nostro caso applicazione perché rombo vuoto) che permette di raggruppare elementi sia atomici che composti.

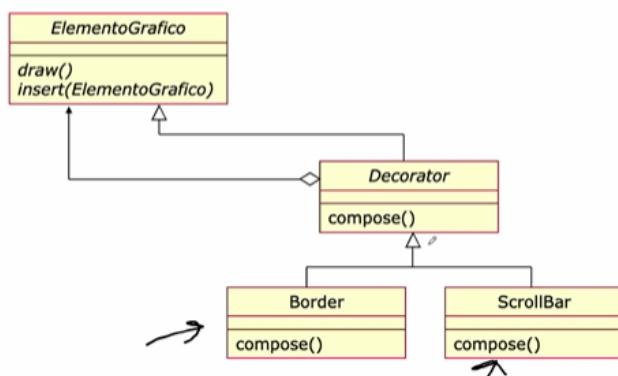
Si ha una classe astratta ElementoGrafico ereditata da Immagine, Testo e Frame con metodo draw() per rappresentare graficamente e insert() per aggiungere un nuovo elemento grafico, chiaramente insert da implementare anche per Immagine e Testo che sono base -> se si utilizza insert su un elemento atomico viene generata un'eccezione.

Rappresentazione degli algoritmi di formattazione: il pattern Strategy



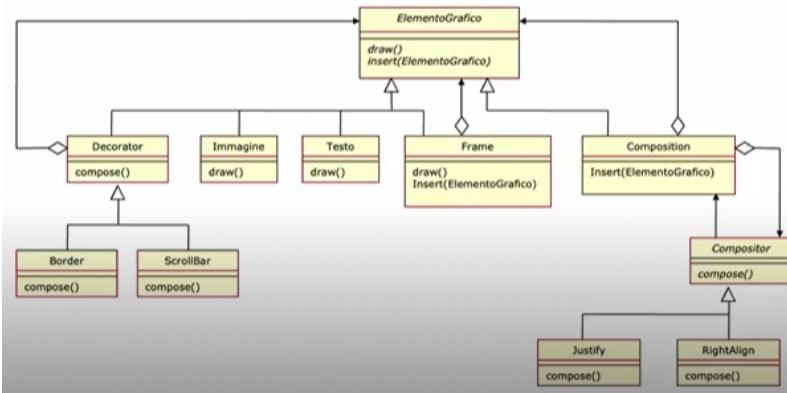
Usiamo strategy per implementare due algoritmi di formattazione: RightAlign e Justify. Quindi poi quando si prende in considerazione un elemento grafico Composto (Composition) secondo quanto visto con Composite l'algoritmo che si utilizza per la formattazione è uno dei due illustrati in figura.

Rappresentazione delle proprietà grafiche: il pattern Decorator



Infine il Pattern Decorator permette di aggiungere Border o ScrollBar all'elemento grafico.

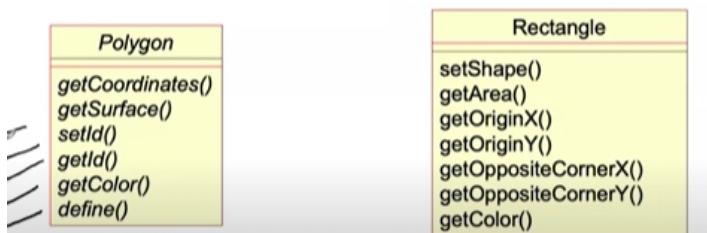
Mettendo tutto insieme...



Nella soluzione complessiva abbiamo l'elemento grafico da cui siamo partiti come classe astratta implementata da Immagine, Testo e Frame che può a sua volta contenere immagine, testo o frame (Composite), Decorator che ha come sottoclassi concrete tutti gli elementi che possono decorare l'elemento grafico e infine Strategy per utilizzare uno dei due algoritmi di formattazione.

Ora vediamo tramite alcuni esempi di applicazione come tradurre i Pattern in codice Java.

Partiamo dall'Adapter, pattern che sappiamo viene usato nel caso in cui si vogliano riusare classi di cui già si dispone ma incompatibili a interfacce che si necessita usare. Si immagina in particolare di voler gestire gli oggetti tramite un'interfaccia `Polygon` e si ha a disposizione una classe `Rectangle` che si vorrebbe riutilizzare, ma che ha interfacce diverse che non si vuole modificare.



(si osserva come alcuni metodi di `Rectangle` conformi all'interfaccia mentre altri no come `getSurface()` non disponibile in `Rectangle`). `setShape()` come costruttore di `Rectangle`.

```

public class Rectangle {
    private float x0, y0;
    private float height, width;
    private String color;
    public void setShape(float x, float y, float a, float l, String c) {
        x0 = x;
        y0 = y;
        height = a;
        width = l;
        color = c;
    }
    public float getArea() {
        return x0 * y0;
    }
    public float getOriginX() {
        return x0;
    }
    public float getOriginY() {
        return y0;
    }
    public float getOppositeCornerX() {
        return x0 + height;
    }
    public float getOppositeCornerY() {
        return y0 + width;
    }
    public String getColor() {
        return color;
    }
}

public interface Polygon {
    public void define(float x0, float y0, float x1, float y1, String color);
    public float[] getCoordinates();
    public float getSurface();
    public void setId(String id);
    public String getId();
    public String getColor();
}

```

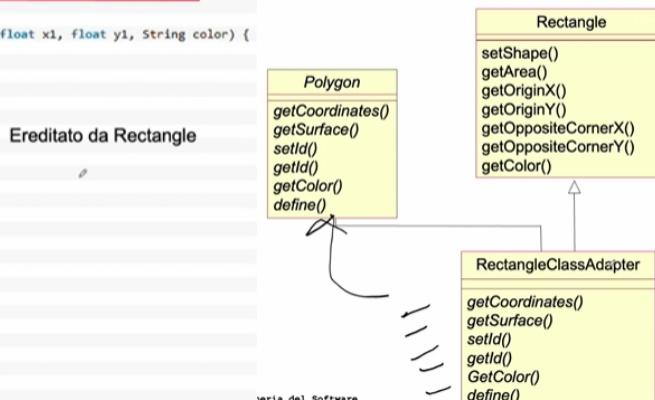
Sappiamo che l'adapter può essere usato sia per oggetti che per classi.

Caso 1) Class Adapter: viene creata una RectangleClassAdapter che estende Rectangle e implementa l'interfaccia Polygon. Essa per implementare l'interfaccia adatterà quindi quanto fornito dalla superclasse.

```

public class RectangleClassAdapter extends Rectangle implements Polygon {
    private String name = "NO NAME";
    public void define(float x0, float y0, float x1, float y1, String color) {
        float a = x1 - x0;
        float l = y1 - y0;
        setShape(x0, y0, a, l, color);
    }
    public float getSurface() {
        return getArea();
    }
    public float[] getCoordinates() {
        float aux[] = new float[4];
        aux[0] = getOriginX();
        aux[1] = getOriginY();
        aux[2] = getOppositeCornerX();
        aux[3] = getOppositeCornerY();
        return aux;
    }
    public void setId(String id) {
        name = id;
    }
    public String getId() {
        return name;
    }
}

```



Ed ecco un possibile client...

```

public class ClassAdapterExample {
    public static void main(String[] args) {
        Polygon block = new RectangleClassAdapter();
        block.setId("Demo");
        block.define(3, 4, 10, 20, "RED");
        System.out.println("The area of " + block.getId() + " is " +
                           + block.getSurface() + ", and it's " + block.getColor());
    }
}

```

Caso 2) Object Adapter: qui invece di estendere la classe da adattare la incapsuliamo in un oggetto. La costruzione dell'Object Adapter si basa quindi sulla creazione di una nuova classe RectangleObjectAdapter che avrà al suo interno un oggetto della classe Rectangle e che implementa l'interfaccia Polygon.

```

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);
    }
    public float getSurface() {
        return adaptee.getArea();
    }
    public float[] getCoordinates() {
        float aux[] = new float[4];
        aux[0] = adaptee.getOriginX();
        aux[1] = adaptee.getOriginY();
        aux[2] = adaptee.getOppositeCornerX();
        aux[3] = adaptee.getOppositeCornerY();
        return aux;
    }
    public void setId(String id) {
        name = id;
    }
    public String getId() {
        return name;
    }
    public String getColor() {
        return adaptee.getColor();
    }
}

```

Si sfrutta l'oggetto adaptee (che è istanza di Rectangle) per adattare i metodi dell'interfaccia richiesti similmente a prima.

Ed ecco un possibile client...

```

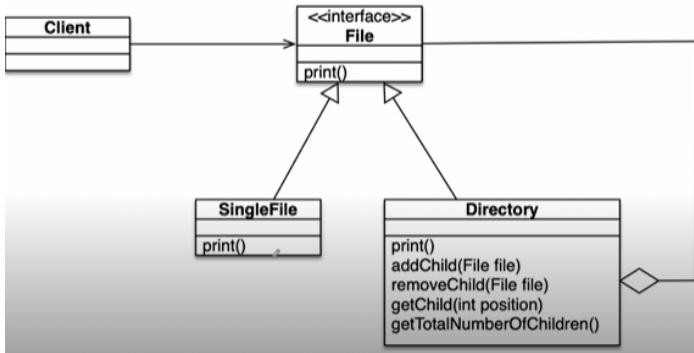
public class ObjectAdapterExample {
    public static void main(String[] args) {
        Polygon block = new RectangleObjectAdapter();
        block.setId("Demo");
        block.define(3, 4, 10, 20, "RED");
        System.out.println("The area of " + block.getId() + " is "
            + block.getSurface() + ", and it's " + block.getColor());
    }
}

```

Vediamo ora un esempio di Composite.

Pensiamo a un FileSystem, struttura tipicamente gerarchica ad albero che può presentare elementi semplici (files) e contenitori (cartelle).

Si vuole permettere al client di accedere e navigare il File System senza conoscere la natura degli elementi che lo compongono in modo da trattarli tutti allo stesso modo. Per far ciò quindi il client userà la stessa interfaccia per l'accesso a entrambi gli elementi, mentre l'implementazione nasconderà la gestione degli oggetti a seconda della loro reale natura.



Il client usa l’interfaccia File tramite cui può stampare il contenuto dell’elemento. Se si parla di un singolo file allora l’implementazione si limita a stampare il singolo file, nel caso delle directory posso stampare l’elenco dei suoi elementi, posso aggiungerci file, rimuoverli, ottenere l’elemento in una certa posizione o ancora ottenere il numero totale di elementi contenuti.

```

public class Directory implements File {
    private final String directoryName;
    private final List<File> children;

    public Directory(String directoryName, List<File> children) {
        this.directoryName = directoryName;
        this.children = new ArrayList<File>(children);
    }

    public void addChild(File file) {
        this.children.add(file);
    }

    public void removeChild(File file) {
        this.children.remove(file);
    }

    public File getChild(int position) {
        if (position < 0 || position >= children.size()) {
            throw new RuntimeException("Invalid position " + position);
        }
        return children.get(position);
    }

    private int getTotalNumberOfChildren() {
        return children.size();
    }

    @Override
    public void print() {
        System.out.println("Printing the contents of the directory - " +
                           directoryName);
        children.forEach(File::print);
        System.out.println("Done printing the contents of the directory - " +
                           directoryName);
    }
}

public class SingleFile implements File {
    private final String fileName;

    public SingleFile(String fileName) {
        this.fileName = fileName;
    }

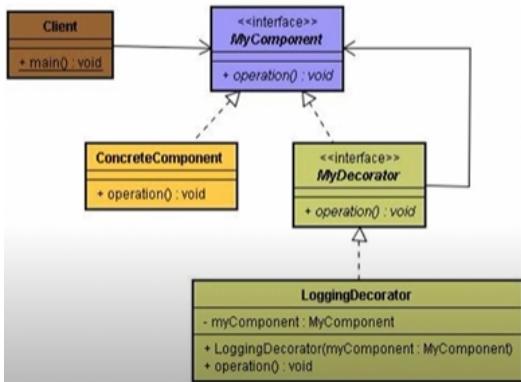
    @Override
    public void print() {
        System.out.println(fileName);
    }
}

```

(poi è esplosa la rec vabbé)

Vediamo un esempio di Decorator, dove consideriamo stavolta uno scenario diverso: abbiamo l’esigenza di monitorare l’invocazione di un metodo senza possibilità di modificare il codice.

Si utilizza quindi il pattern Decorator per esigenze di debug; “wrappiamo” un metodo con delle semplici istruzioni print-screen.



L'interfaccia Component fornisce una certa operazione implementata da ConcreteComponent (questa operazione la dobbiamo monitorare), a questo punto si introduce l'interfaccia MyDecorator che estende Component ed è implementata dal LoggingDecorator, che aggiunge l'operazione LoggingDecorator e attraverso la quale viene creato l'oggetto decoratore che contiene l'oggetto da decorare.

```

package patterns.decorator;
package patterns.decorator;
public interface MyComponent {
    public void operation();
}
public class ConcreteComponent implements MyComponent {
    public void operation(){
        System.out.println("Hello World");
    }
}
  
```

Ipotizzando di non poter usare il codice di ConcreteComponent per monitorare il metodo operation, utilizziamo il Decoratore definendo l'interfaccia Decorator e la classe LoggingDecorator.

```

public class LoggingDecorator implements MyDecorator {
    MyComponent myComponent = null;
    public LoggingDecorator(MyComponent myComponent){
        this.myComponent = myComponent;
    }
    public void operation() {
        System.out.println("First Logging");
        myComponent.operation();
        System.out.println("Last Logging");
    }
}
  
```

Definiamo l'interfaccia MyDecorator che si occupa di ereditare il metodo interessato da MyComponent e di interporci con le classi di decorazione concrete.

Creiamo la classe LoggingDecorator che implementa l'interfaccia MyDecorator ed aggiunge le informazioni di debug prima e dopo l'esecuzione del metodo interessato.

Quando si crea l'oggetto decoratore si assegna l'attributo myComponent ed ogni volta che si esegue l'operazione stampa a video dei messaggi che monitorano l'esecuzione del metodo.

```

class Client {
    public static void main(String[] args) {
        MyComponent myComponent = new LoggingDecorator(new ConcreteComponent());
        myComponent.operation();
    }
}
  
```

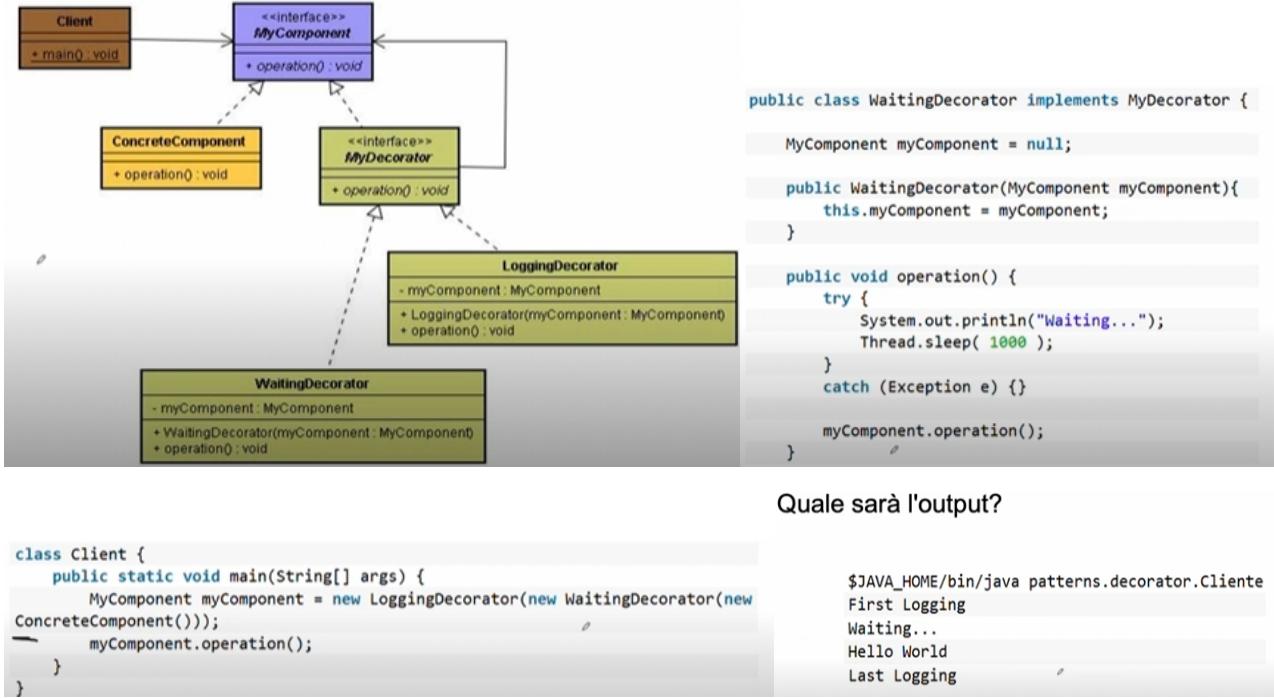
La classe Client invoca la classe concreta LoggingDecorator passando al costruttore il componente concreto, successivamente invoca il metodo operation().

Quale sarà l'output?

```

$JAVA_HOME/bin/java patterns.decorator.Cliente
First Logging
Hello World
Last Logging
  
```

Partendo dall'esempio possiamo creare diverse classi concrete Decorator che aggiungano nuove funzionalità, come una classe WaitingDecorator che preveda una pausa durante l'esecuzione. Vediamo come diventa il Class Diagram in seguito all'inserimento di questa classe.



Ora vediamo un esempio di Factory Method.

Si suppone di avere un'applicazione che legge dati da un file di testo contenente informazioni relative a rilevazioni di letture di contatori per acqua e gas.

Abbiamo nel nostro codice una classe dedicata a ciò che legge i vari formati di file: `AcquisizioneLetture`.

```

public class AcquisizioneLetture {
    public AcquisizioneLetture() {
    }

    public void parseFile(String fileName, String dataType) {
        ArrayList letturaArray = new ArrayList();
        FileLetturaReader fileLetturaReader; //questa è una interfaccia
        Lettura lettura; //questa è una interfaccia

        if (dataType.equals("gas")) {
            fileLetturaReader = new GasLetturaReader(fileName);
        }
        if (dataType.equals("H2O")) {
            fileLetturaReader = new H2OLetturaReader(fileName);
        }

        while (fileLetturaReader.hasNextLettura()) {
            lettura = fileLetturaReader.getNextLettura();

            if (lettura.verifica()) {
                lettura.calcolaconsumo();
                lettura.registra();
            } else {
                lettura.scarta();
            }
        }
    }
}

```

Questa implementazione è corretta ma poco flessibile: immaginiamo ad es. che il client inizia a vendere anche energia elettrica e quindi acquisire anche i file con le relative letture. Sarà necessario modificare la classe per gestire questo tipo aggiuntivo di file.

```
if(dataType.equals("EE")) {  
    fileLetturaReader = new EELetturaReader(fileName);  
}
```

Conviene quindi usare il pattern Factory Method, che insieme all'Abstract Factory è Pattern creazionale. L'obiettivo è separare il codice soggetto a modifiche dalla restante parte che resta sempre uguale, come fare?

Un'idea è encapsulare la creazione di FileLetturaReader in una nuova classe FileReaderFactory:

```
public class ReaderFactory {  
    private FileLetturaReader fileLetturaReader;  
    public ReaderFactory() {  
    }  
  
    public FileLetturaReader getFileLetturaReader(String fileName, String dataType) {  
        if (dataType.equals("gas")) {  
            fileLetturaReader = new GasLetturaReader(fileName);  
        }  
        if (dataType.equals("H2O")) {  
            fileLetturaReader = new H2OLetturaReader(fileName);  
        }  
        if (dataType.equals("EE")) {  
            fileLetturaReader = new EELetturaReader(fileName);  
        }  
        return fileLetturaReader;  
    }  
}
```

Si modifica di conseguenza AcquisizioneLetture affinché sia responsabilità della factory gestire la logica di lettura del documento e non del client.

```
public class AcquisizioneLetture {  
    private ReaderFactory factory;  
    public AcquisizioneLetture(ReaderFactory factory) {  
        this.factory = factory;  
    }  
  
    public void parseFile(String fileName, String dataType) {  
        ArrayList letturaArray = new ArrayList();  
        FileLetturaReader fileLetturaReader; //questa è una interfaccia  
        Lettura lettura; //questa è una interfaccia  
        // meglio, no?  
        fileLetturaReader = factory.getFileLetturaReader(fileName, dataType);  
        //  
        while (fileLetturaReader.hasNextLettura()) {  
            lettura = fileLetturaReader.getNextLettura();  
  
            if (lettura.verifica()) {  
                lettura.calcolaConsumo();  
                lettura.registra();  
            } else {  
                lettura.scarta();  
            }  
        }  
    }  
}
```

In questo modo la classe AcquisizioneLetture non dovrà più esser modificata in caso di nuovi tipi di file da leggere.

Ora ipotizziamo che acquisiamo due importanti clienti, che vendono acqua e gas utilizzando però formati XML differenti.

Adeguiamo quindi il codice alle nuove esigenze scrivendo due nuove factory Cliente1ReaderFactory e Cliente2ReaderFactory che derivano dalla nostra ReaderFactory.

```

public class Cliente1ReaderFactory extends ReaderFactory{
    public Cliente1ReaderFactory() {
    }

    public FileLetturaReader getFileLetturaReader(String fileName, String dataType){
        if (dataType.equals("gas")){
            fileLetturaReader = new Cliente1GasLetturaReader(fileName);
        }
        if (dataType.equals("H2O")){
            fileLetturaReader = new Cliente1H2OLetturaReader(fileName);
        }
        return fileLetturaReader;
    }
}

public class Cliente2ReaderFactory extends ReaderFactory {
    public Cliente2ReaderFactory() {
    }

    public FileLetturaReader getFileLetturaReader(String fileName,
                                                String dataType){
        if (dataType.equals("gas")){
            fileLetturaReader = new Cliente2GasLetturaReader(fileName);
        }
        if (dataType.equals("H2O")){
            fileLetturaReader = new Cliente2H2OLetturaReader(fileName);
        }
        return fileLetturaReader;
    }
}

```

L'unica differenza tra i due codici è che si usano in uno il lettore specifico per il file XML del client 1 e il secondo per il client 2.

Possiamo quindi migliorare il codice portando il Factory direttamente in AcquisizioneLettura per rendere la classe autosufficiente, senza che però perda la flessibilità ottenuta fino ad ora.

```

public abstract class AcquisizioneLettura {
    public AcquisizioneLettura() {
    }

    public void parseFile(String fileName, String dataType) {
        ArrayList letturaArray = new ArrayList();
        FileLetturaReader fileLetturaReader; //questa è una interfaccia
        Lettura lettura; //questa è una interfaccia
        fileLetturaReader = getFileLetturaReader(fileName, dataType);
        //while (fileLetturaReader.hasNextLettura()) {
        //    lettura = fileLetturaReader.getNextLettura();
        //    if (lettura.verifica()) {
        //        lettura.calcolaConsumo();
        //        lettura.registra();
        //    } else {
        //        lettura.scarta();
        //    }
        //}
        protected abstract FileLetturaReader getFileLetturaReader(String fileName, String fileType);
    }
}

public class AcquisizioneLettureConcreta extends AcquisizioneLettura {
    public AcquisizioneLettureConcreta() {
    }

    protected FileLetturaReader getFileLetturaReader(String fileName,
                                                    String fileType) {
        FileLetturaReader fileLetturaReader;
        if (fileType.equals("gas")){
            fileLetturaReader = new GasLetturaReader(fileName);
        }
        if (fileType.equals("H2O")){
            fileLetturaReader = new H2OLetturaReader(fileName);
        }
        if (fileType.equals("EE")){
            fileLetturaReader = new EELetturaReader(fileName);
        }
        return fileLetturaReader;
    }
}

public class AcquisizioneLettureCliente1 extends AcquisizioneLettura {
    public AcquisizioneLettureCliente1() {
    }

    protected FileLetturaReader getFileLetturaReader(String fileName,
                                                    String fileType) {
        FileLetturaReader fileLetturaReader;
        if (fileType.equals("gas")){
            fileLetturaReader = new Cliente1GasLetturaReader(fileName);
        }
        if (fileType.equals("H2O")){
            fileLetturaReader = new Cliente1H2OLetturaReader(fileName);
        }
        return fileLetturaReader;
    }
}

public class AcquisizioneLettureCliente2 extends AcquisizioneLettura {
    public AcquisizioneLettureCliente2() {
    }

    protected FileLetturaReader getFileLetturaReader(String fileName,
                                                    String fileType) {
        FileLetturaReader fileLetturaReader;
        if (fileType.equals("gas")){
            fileLetturaReader = new Cliente2GasLetturaReader(fileName);
        }
        if (fileType.equals("H2O")){
            fileLetturaReader = new Cliente2H2OLetturaReader(fileName);
        }
        return fileLetturaReader;
    }
}

```

La classe base non conosce il FileLetturaReader su cui itera e da cui ricava le lettura, perché questo dipende dalle classi derivate. Abbiamo conservato disaccoppiamento e flessibilità.

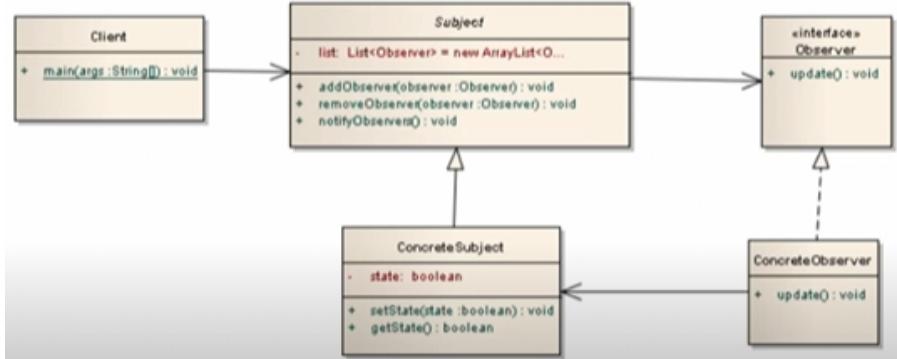
CHE SCHIFO

Vediamo infine un esempio per l'observer.

Due osservatori sono interessati al cambio di stato di un soggetto osservato.

Successivamente uno dei due cancella la sottoscrizione e non riceve più notifiche.

In questo caso dobbiamo creare il Subject e l'Observer e le classi concrete rispettive.



```
public abstract class Subject {
    private List<Observer> list = new ArrayList<Observer>();

    public void addObserver(Observer observer) {
        list.add( observer );
    }

    public void removeObserver(Observer observer) {
        list.remove( observer );
    }

    public void notifyObservers() {
        for(Observer observer: list) {
            observer.update();
        }
    }
}
```

Il Subject include una lista degli osservatori che si registrano presso il soggetto osservato tramite i metodi `addObserver()` e si cancellano tramite il metodo `removeObserver()`. Mentre invece il metodo `notifyObservers()` viene invocato dalla classe concreta `ConcreteSubject` quando interviene un cambio di stato.

```
public class ConcreteObserver implements Observer {
    @Override
    public void update() {
        System.out.println("Sono " + this + ": il Subject e' stato modificato!");
    }
}
```

Il `ConcreteObserver` implementa il metodo `update()` per definire l'azione da intraprendere quando interviene un cambio di stato del `Subject`.

Esplosa la rec dio can

