- ***Lezione 26*** - Esercizio

- ***Lezione 27***

➔ *Decimo Blocco*

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

1. 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

2. 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
Raggio d'azione	Classi	**Factory Method**	**Adapter** (class)	Interpreter **Template Method**
	Oggetti	**Abstract Factory** Builder Prototype Singleton	**Adapter** (object) Bridge **Composite** **Decorator** Facade Flyweight Proxy	Chain of responsability 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

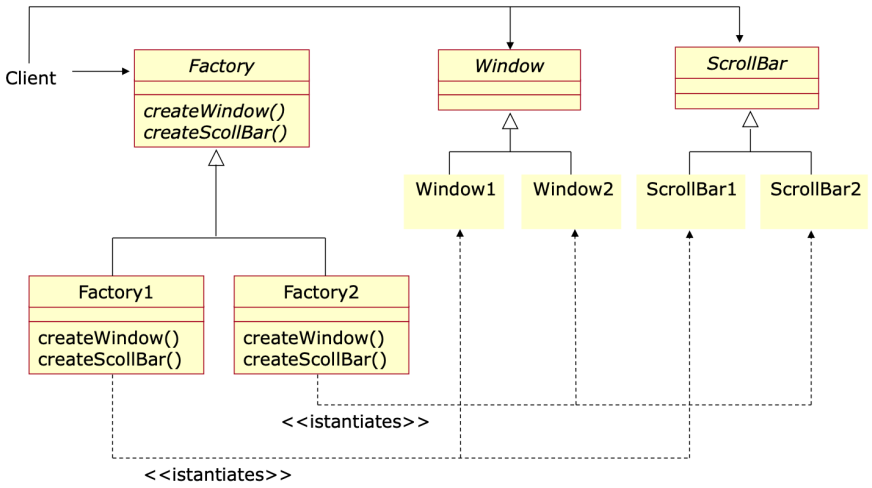
Ogni pattern che descriveremo quindi sarà presentato con questo formato.
Prima di passare ai pattern ricordiamo un concetto utile, i Framework.
Un Framework non rappresenta una semplice libreria, ma un insieme di classi che cooperano al fine di fornire lo scheletro di un'applicazione riusabile in uno 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 permettono quindi di definire lo scopo e la struttura statica di un sistema, permettendo di raggiungere gli obiettivi di riuso di design e di riuso del codice (anche se principalmente a livello di design (progetto) in quanto definisce lo scheletro di interazione tra oggetti).
Quando si utilizza una libreria di classi/funzioni ciò che si fa è invocare quando necessario classi/funzioni messe a disposizione dalla libreria, quando si usa il framework invece si avrà a disposizione un corpo principale e si scrive il codice che verrà chiamato da questo corpo principale (quindi rappresenta un po' l'inverso rispetto al riuso a cui siamo abituati, dove si chiamano funzioni/classi scritte da altri).

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.
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 architetturale 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 tra loro che devono essere creati in modo combinato.
  - Motivazione: si suppone di voler realizzare un software la cui interfaccia utente deve saper supportare diversi tipi di look & feel (ossia gli elementi che fanno parte dell'interfaccia come finestre, menù etc.. sono sempre gli stessi, ma si vogliono avere diversi modi di visualizzarli es. versione chiara/scura). Per garantire la portabilità di un'applicazione tra look & feel diversi, gli oggetti non devono essere cablati nel codice (se ogni oggetto grafico come bottoni, finestre, menù è creato direttamente nel codice con tipo specifico, sarà difficile cambiare stile in futuro!).

Esempio_Abstract_Factory: 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.
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.

Il rispetto delle relazioni è cablato nel codice e deve essere noto al client.

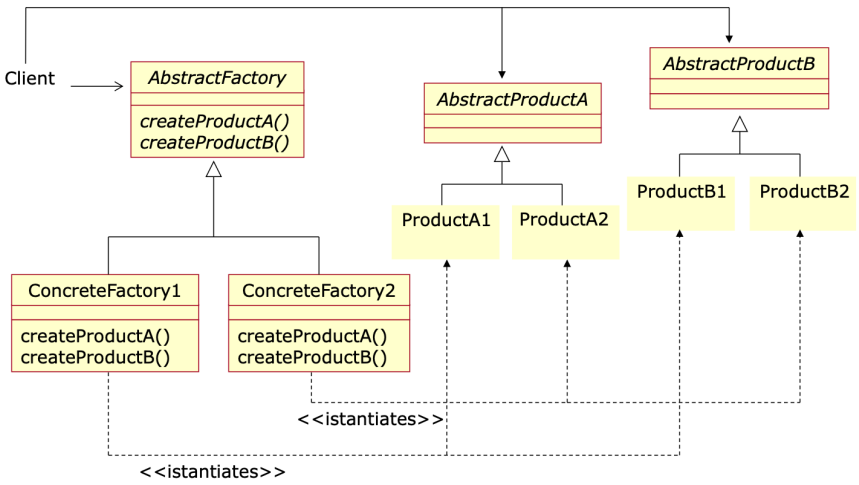
```
Window w = new Window1();
...
ScrollBar s = new ScrollBar1();
```

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:*



Tornando alle caratteristiche standard dell'abstract factory:

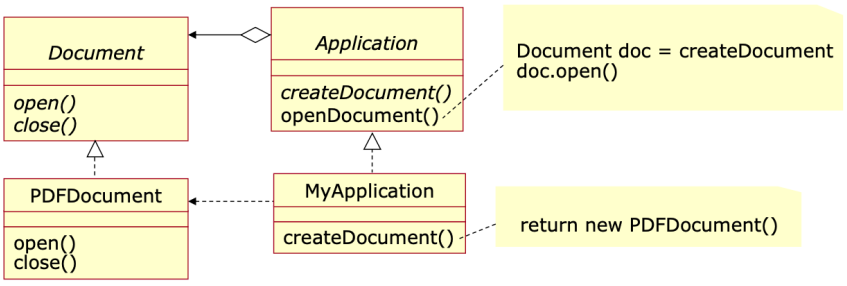
- *Applicabilità:* si fa riferimento a software indipendente dalle modalità di creazione dei prodotti con cui opera (il client non deve sapere come vengono creati gli oggetti), in particolare è utile quando il sistema è configurabile con famiglie di prodotti diverse ed il client non è legato a una specifica 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).

- Factory Method → ha sempre uno scopo creazionale, ma stavolta lo scope non è su oggetti ma su classi:

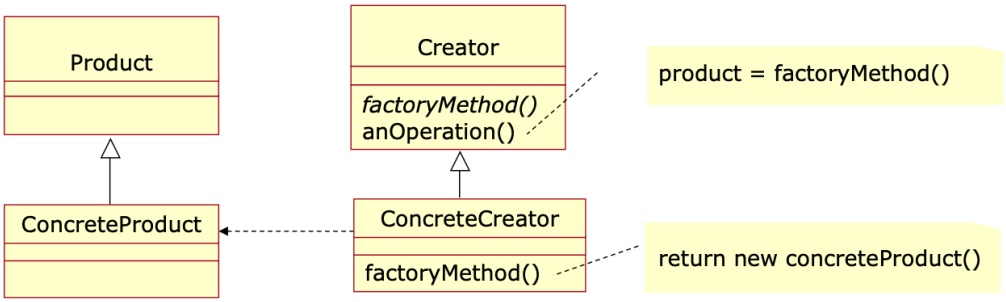
  - *Scopo:* definire un'interfaccia per la creazione di un oggetto, che consenta di decidere a tempo di esecuzione quale specifico oggetto istanziare
  (mentre quindi nell'abstract factory l'ereditarietà è utilizzata in modo statico (aggiungere una nuova factory richiede ricompilazione), qui si **delega** invece il compito runtime (dinamico) a una sottoclasse).
  (Il Factory Method permette a una classe di delegare la creazione degli oggetti a sottoclassi, in modo che il tipo concreto dell'oggetto creato possa variare senza modificare il codice principale)
  - *Motivazione:* ampiamente utilizzato nei framework, dove si hanno classi astratte che definiscono le relazioni tra elementi del dominio e sono poi responsabili per la creazione degli oggetti concreti.

  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 (può crearlo in qualsiasi momento) ma non sa esattamente quale deve creare (questo è delegato alle sottoclassi definite dagli utilizzatori del framework).

  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.



- *Struttura:*



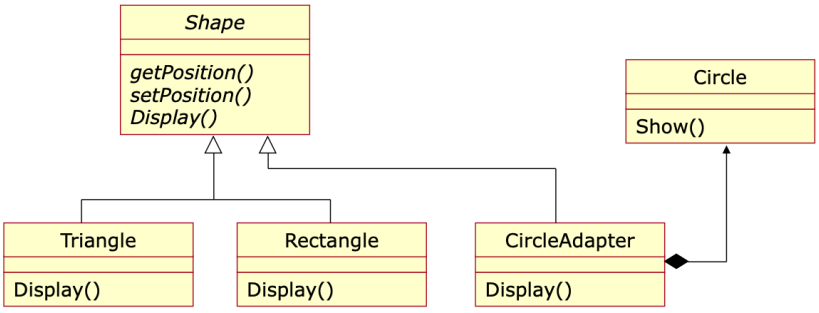
In generale l'operazione factoryMethod() (che fa parte della classe astratta Creator delegherà la creazione dell'oggetto del prodotto specifico a una sottoclasse realizzata dagli utilizzatori del framework (qui si vede chiaramente il riuso "inverso" dei framework per cui non chiamo un metodo che mi è 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).

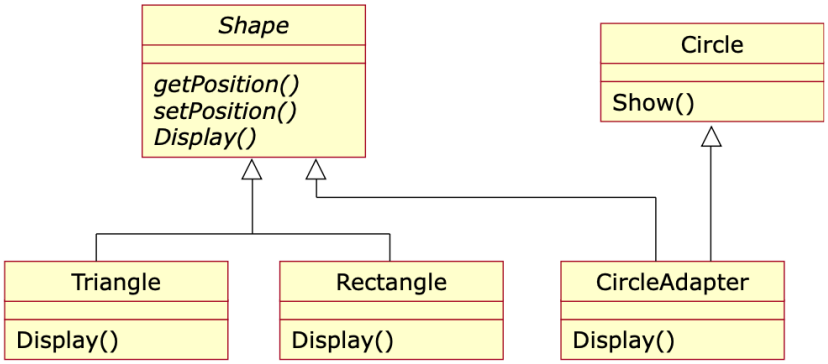
- Pattern Adapter, pattern di scopo Strutturale unico che può essere applicato sia con scope Object che con scope Class.

  - *Scope:* 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

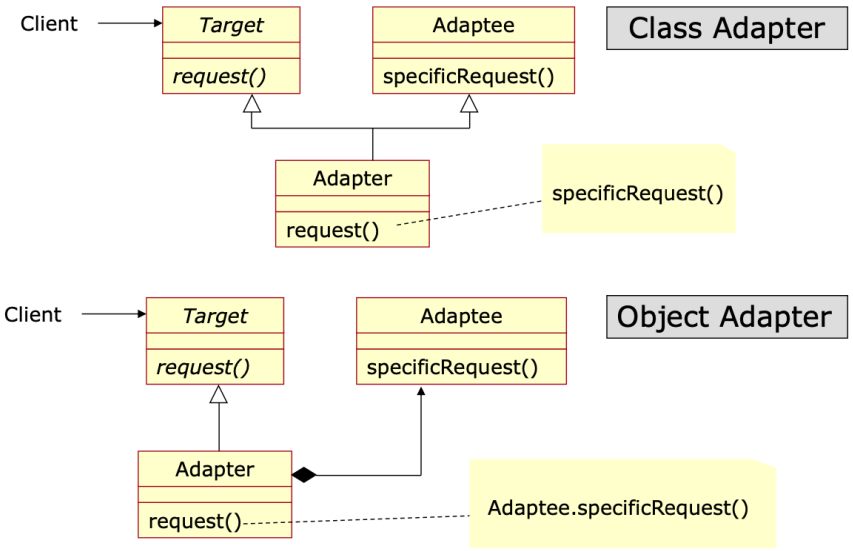


Si realizza una classe chiamata non Circle ma CircleAdapter che crea l'oggetto Circle (riusando la classe Circle), e quando viene invocato il metodo Display() nell'adattatore sarà invocato il relativo metodo Show() per Circle.
Si utilizza la Composition UML (rombo pieno) in quanto appunto l'oggetto Circle è creato all'interno dell'oggetto CircleAdapter → è incapsulato.
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() ridefiniscei il metodo Show() facente parte della classe Circle.

- *Struttura:*

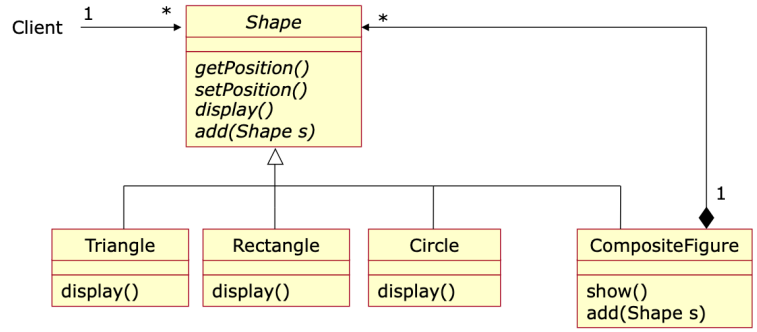


Si ha l'interfaccia Target a cui dobbiamo adattarci, la classe Adapter adattatore e Adaptee da adattare. Con Class Adapter si fa uso dell'ereditarietà (si chiama il metodo della superclasse), con Object Adapter composizione (si crea l'oggetto dall'Adapter su cui viene invocato il metodo specificRequest()).

- *Adattabilità:* 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).

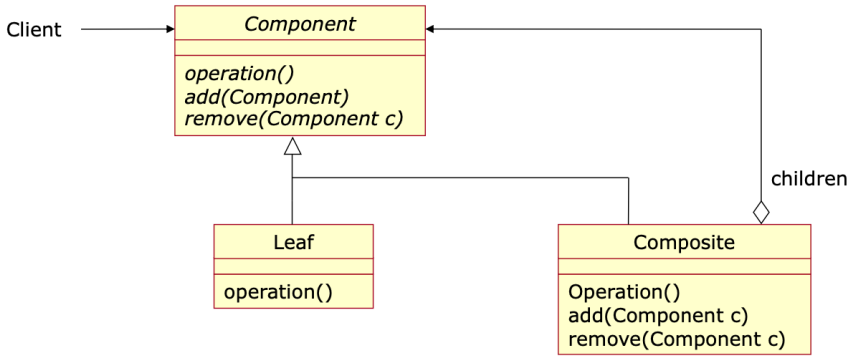
- Composite, che ha purpose strutturale e scope basato su oggetti.

  - *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 compositi. 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 (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:*



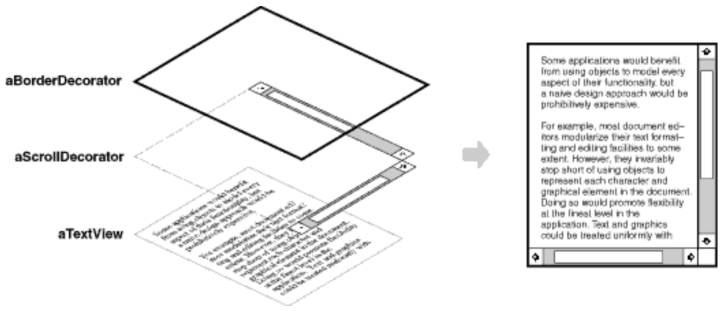
Il Client interagisce con il componente generico avente una certa interfaccia, classi elementari (Leaf) che costituiscono direttamente Component (possono essere quindi più di uno) e Composite ricordandomi di usare non solo il costrutto di aggregazione o composizione ma anche ereditarietà.
Se si usa composition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

- ***Lezione 28***

- Il Decorator. Si tratta di un pattern con purpose strutturale e scope basato su oggetti.

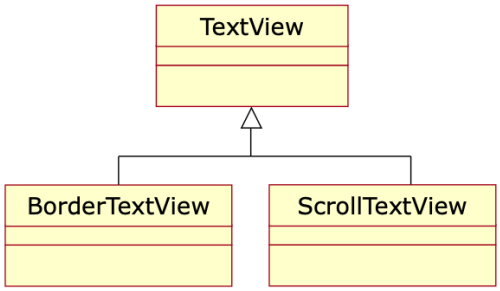
  - *Scopo:* si vogliono "decorare" gli oggetti. Una decorazione rappresenta l'aggiungere dinamicamente (runtime) funzionalità (responsabilità) ad un oggetto. Il subclassing (per cui a partire da una classe generale ne definisco di più specifiche che ereditano tutto e in più aggiungono funzionalità) rappresenta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



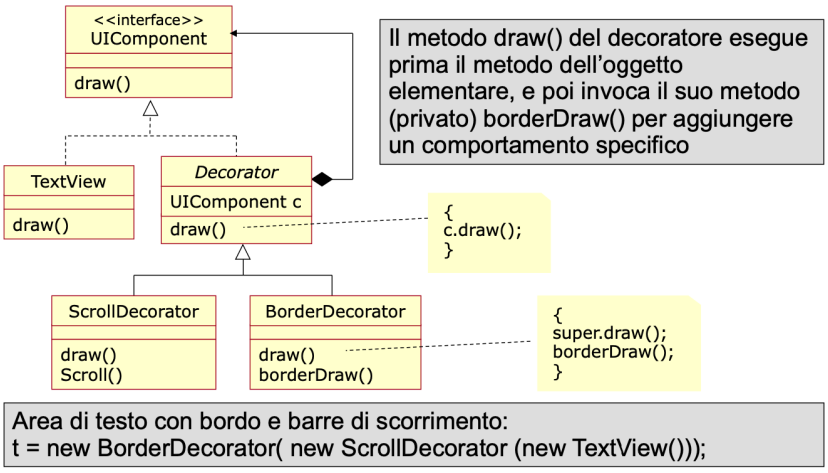
Come combinare i tre oggetti aBorderDecorator, aScrollDecorator e aTextView per raggiungere l'obiettivo della dinamicità?

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



Estendendo la classe TextView con la classe BorderTextView, che si occupa di andare ad aggiungere un brodo all'oggetto classico TextView;

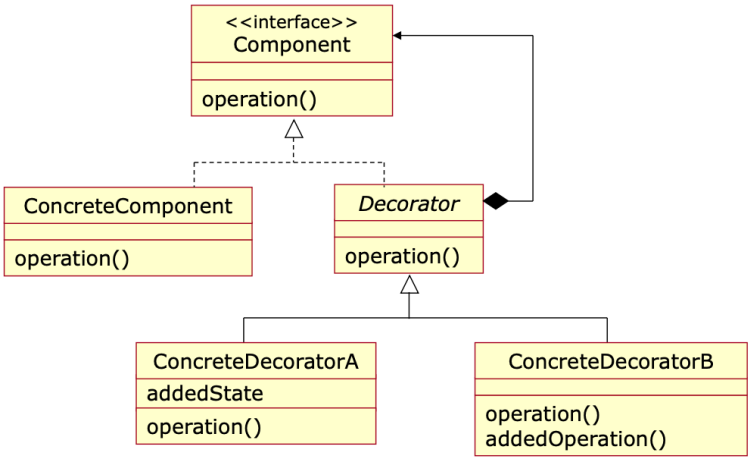
A questo punto facciamo nuovamente riferimento al costrutto di Composizione.
L'idea di decorare un oggetto, ossia aggiungerci funzionalità, può essere realizzata infatti in modo molto più flessibile racchiudendo un oggetto elementare in un altro che aggiunge una responsabilità particolare (l'oggetto più specifico incapsula l'oggetto base, e in questo senso l'oggetto contenitore che contiene l'oggetto da decorare è chiamato Decorator).
Il Decorator ovviamente deve quindi avere un'interfaccia conforme all'oggetto da decorare, ciò che succede è quindi che una volta incapsulato l'oggetto base il decorator trasferisce le richieste all'oggetto decorato ma può svolgere funzioni aggiuntive (come appunto aggiungere un bordo o la scrollbar) prima o dopo il trasferimento della richiesta.
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 contiene UIComponent → può contenere un oggetto TextView.



Tuttavia Decorator rappresenta ancora una classe astratta → da essa eredito ScrollDecorator e BorderDecorator, in particolare ognuna di esse aggiunge il metodo necessario per estendere le funzionalità base dell'oggetto elementare (TextView).
In questo caso il metodo draw() esegue prima il metodo dell'oggetto elementare, e poi chiama eventualmente il suo metodo 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) → 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:*



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.

- *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:

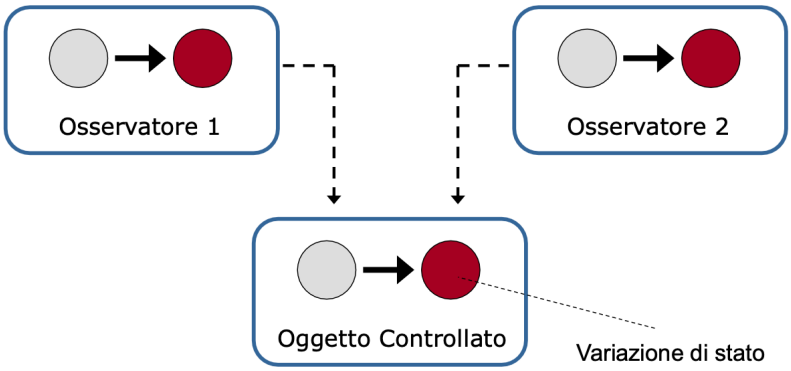
```
BufferedInputStream bin = new BufferedInputStream( new FileInputStream( "test.dat"));
```

(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.

- 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 da 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 itnerfaccia grafica secondo il paradigma Model-View-Control (== 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.

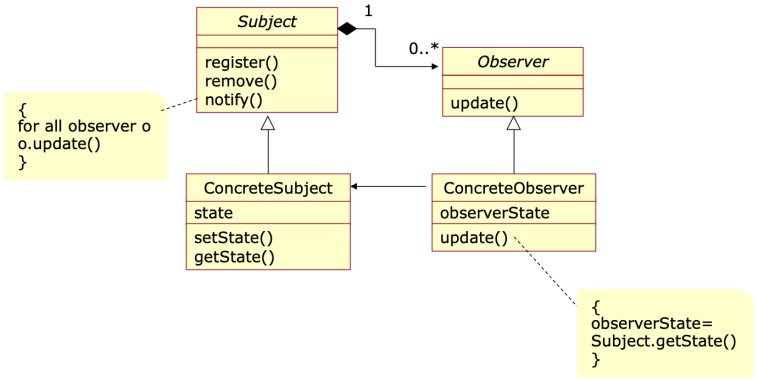


Esempio: software che mostra i risultati di una partita di calcio, gli oggetti legati al risultato possono cambiare dinamicamente stato in tempi anche molto ristretti, si vuole fare in modo che quando lo stato cambia gli osservatori (come le semplici interfacce grafiche che mostrano all'utente questi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

- ***Lezione 29***

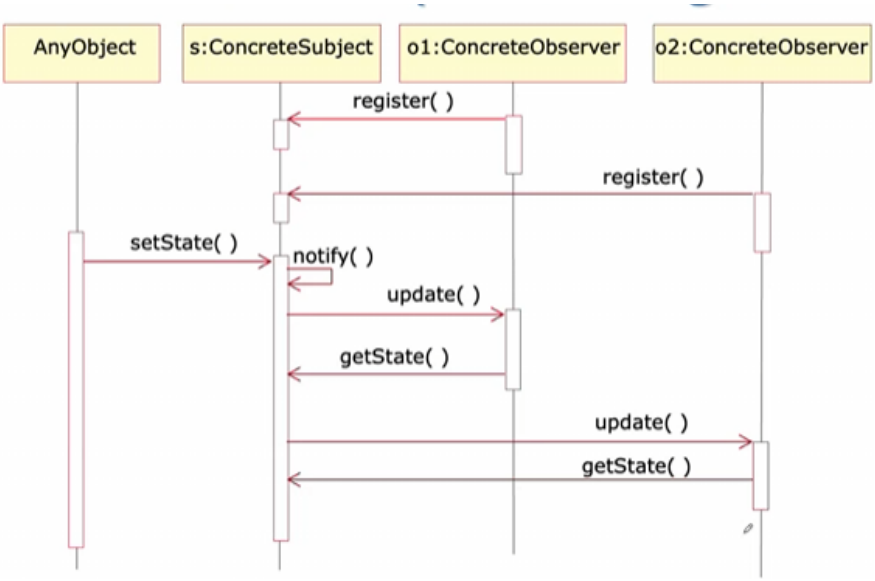
L'approccio corretto è quello per cui invece sia l'oggetto osservato a dover notificare gli osservatori in caso di cambiamento.
Il pattern Observer in particolare prevede che gli osservatori si registrino presso l'oggetto osservato, in questo modo sarà lui a notificare ogni cambiamento di stato agli osservatori. Quando l'osservatore rileva la notifica, può interrogare l'oggetto osservato oppure svolgere operazioni indipendenti dallo specifico stato.

- 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 invece il metodo eseguito dall'observer una volta ricevuta la notifica.
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:

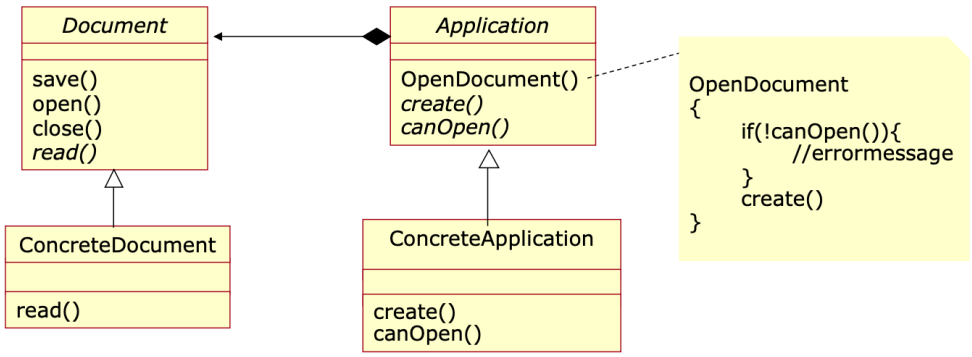


s, o1, o2 rappresentano istanze. Ogni altro oggetto dell'applicazione chiama setState() sull'oggetto osservato facendogli quindi cambiare stato. Quindi il concretesubject invoca su se stesso il metodo notify() che poi invoca l'update su tutti gli oggetti che si erano registrati. Dopodiché l'osservatore,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

- *Applicabilità:* molto utile per mantenere basso il livello di coupling (la soluzione naive 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.

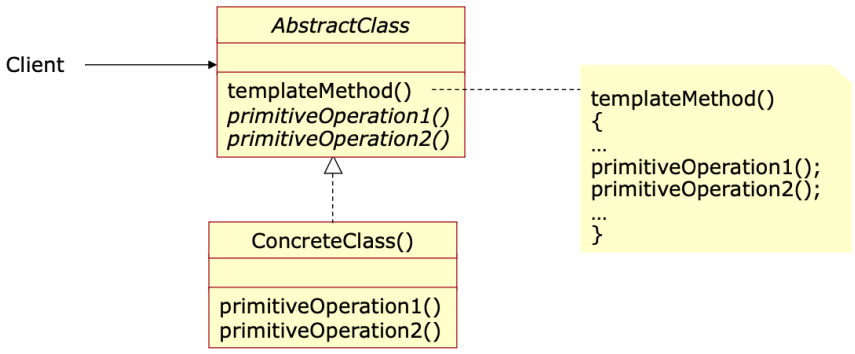
Template Method è un Pattern comportamentale basato su classi.

- *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 considera un framework per costruire applicazioni in base di gestire diversi documenti. Il Template Method definisce un algoritmo in base ad operazioni astratte che saranno definite nelle sottoclassi specifiche



Document e Application classi astratte, il primo prevede i metodi salva, apri e chiudi mentre read() che cambia in base al tipo di documento → se devo leggere Word devo implementare l'algoritmo necessario a leggerlo etc…
In Application invece metodi create, canopen e OpenDocument() che definisce l'algoritmo necessario per creare e leggere il documento tramite i primi due (il corpo prevede che se non si può aprire allora errore altrimenti crealo.
Tuttavia i metodi create() e canOpen() sono astratti e implementati in ConcreteApplication in base al tipo di documento (struttura di controllo invertito!).

- *Struttura:*

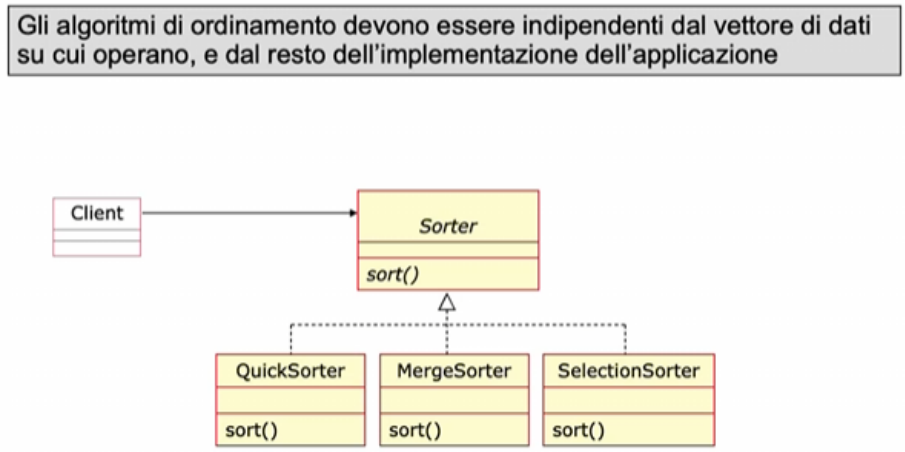


Il client invoca il metodo template che prevede una serie di operazioni, alcune delle quali però sono implementate da sottoclassi che concretizzano la classe astratta.
- *Applicabilità:* utilizzato per implementare la parte invariante di un algoritmo, lasciando alle sottoclassi la definizione degli step 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.

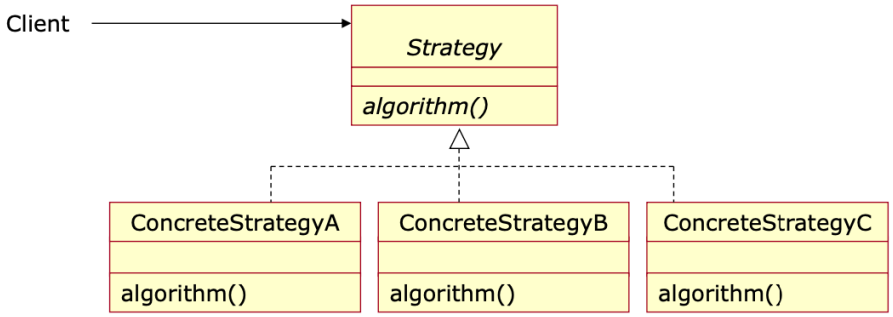
Ci accorgiamo che TemplateMethod e FactoryMethod sono simili tra loro nell'approccio utilizzato, quali sono le differenze?
Infatti il FactoryMethod è un Pattern di tipo creazionale che si basa anch'esso sul delegare la responsabilità di creazione di un oggetto alle sottoclassi.
La differenza principale 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.



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



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 comoportamenti in un’unica classe.
Lo svantaggio principale sta nel fatto che i Client devono conoscere le diverse strategie.