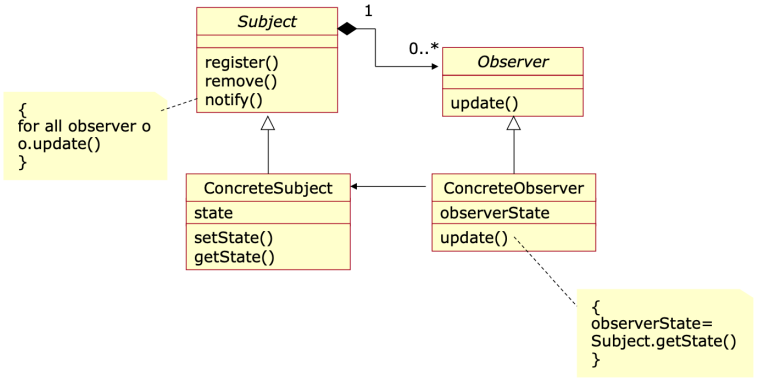
- ***Lezione 29***

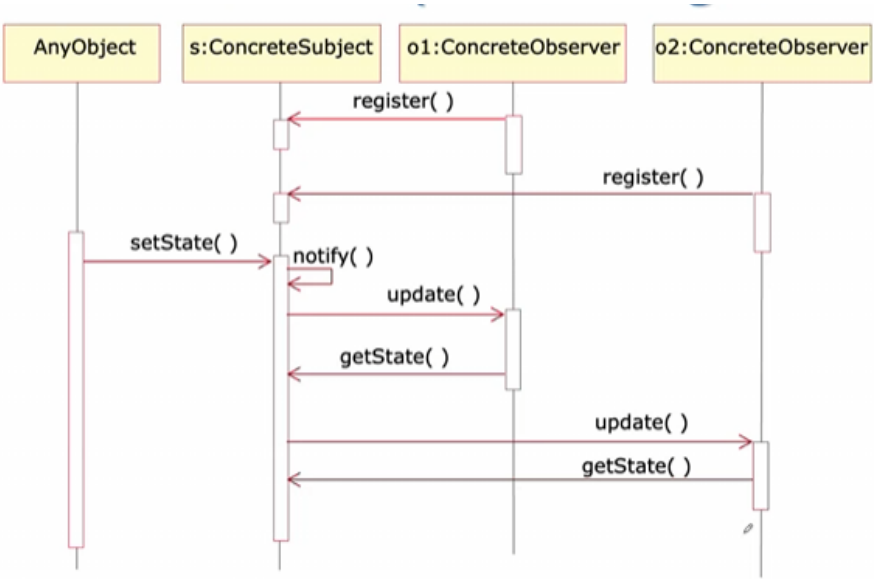
L'approccio corretto è quello per cui invece sia l'oggetto osservato a dover notificare gli osservatori in caso di cambiamento.
Il pattern Observer in particolare prevede che gli osservatori si registrino presso l'oggetto osservato, in questo modo sarà lui a notificare ogni cambiamento di stato agli 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:

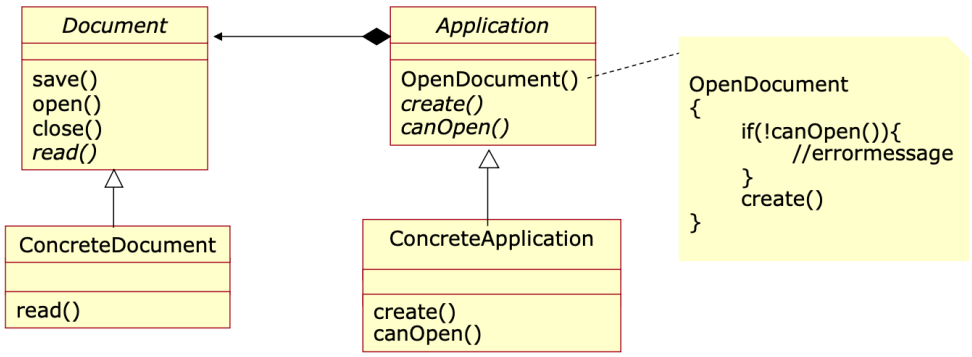


s, o1, o2 rappresentano istanze. Ogni altro oggetto dell'applicazione chiama setState() sull'oggetto osservato facendogli quindi cambiare stato. Quindi il concretesubject invoca su se stesso il metodo notify() che poi invoca l'update su tutti gli oggetti che si erano registrati. Dopodiché l'osservatore,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

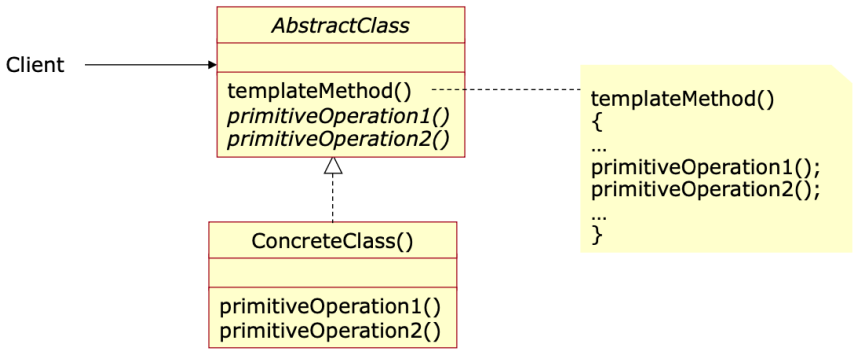
Template Method è un Pattern comportamentale basato su classi.

- *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:*

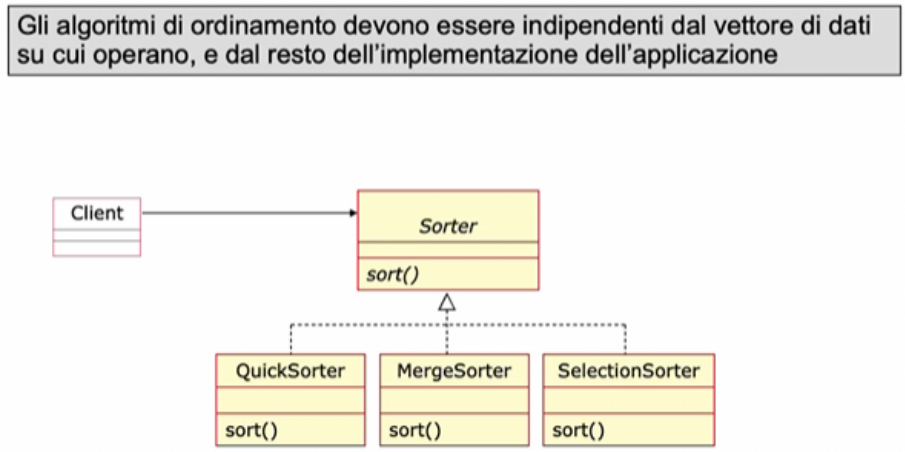


Il client invoca il metodo template che prevede una serie di operazioni, alcune delle quali però sono implementate da sottoclassi che concretizzano la classe astratta.
- *Applicabilità:* utilizzato per implementare la parte invariante di un algoritmo, lasciando alle sottoclassi la definizione degli step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

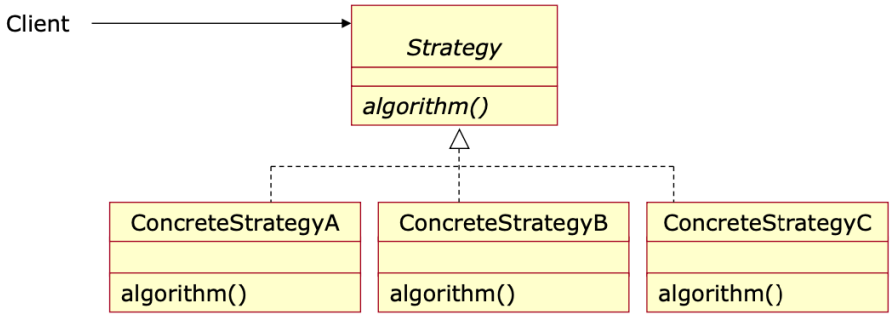
Ci accorgiamo che TemplateMethod e FactoryMethod sono simili tra loro nell'approccio utilizzato, quali sono le differenze?
Infatti il FactoryMethod è un Pattern di tipo creazionale che si basa anch'esso sul delegare la responsabilità di creazione di un oggetto alle sottoclassi.
La differenza principale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