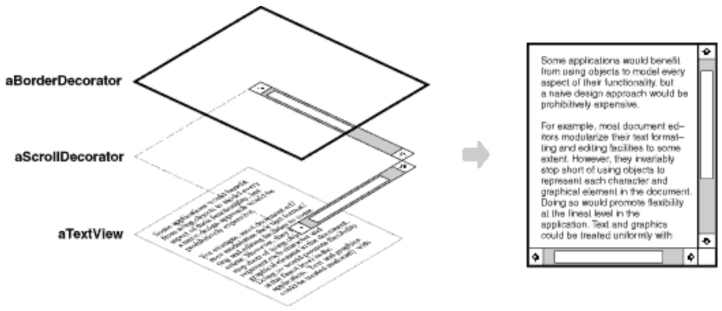
- ***Lezione 28***

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

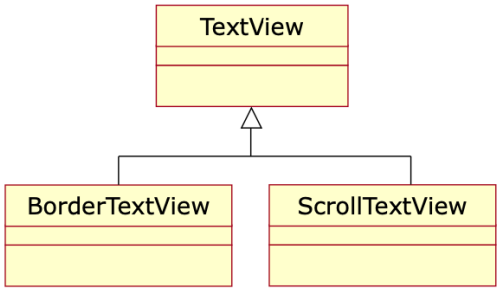
  - *Scopo:* si vogliono "decorare" gli oggetti. Una decorazione rappresenta l'aggiungere dinamicamente (runtime) funzionalità (responsabilità) ad un oggetto. Il subclassing (per cui a partire da una classe generale ne definisco di più specifiche che ereditano tutto e in più aggiungono funzionalità) rappresenta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



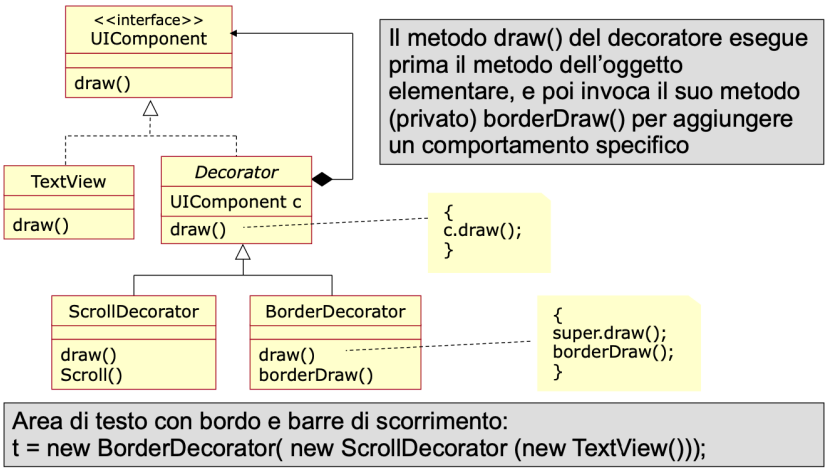
Come combinare i tre oggetti aBorderDecorator, aScrollDecorator e aTextView per raggiungere l'obiettivo della dinamicità?

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;

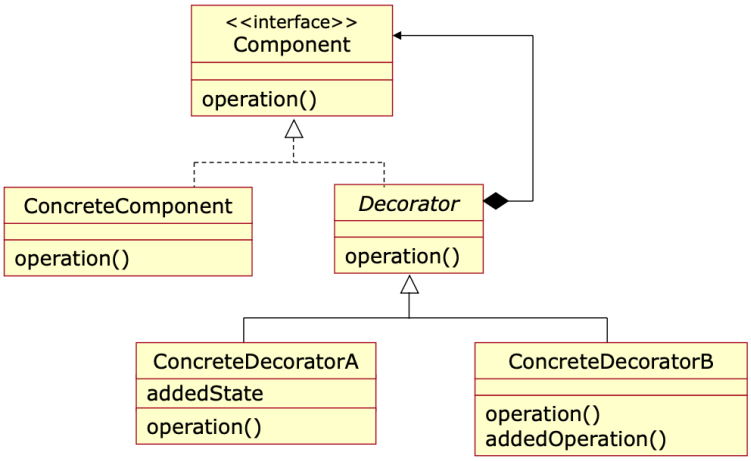
A questo punto facciamo nuovamente riferimento al costrutto di Composizione.
L'idea di decorare un oggetto, ossia aggiungerci funzionalità, può essere realizzata infatti in modo molto più flessibile racchiudendo un oggetto elementare in un altro che aggiunge una responsabilità particolare (l'oggetto 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:

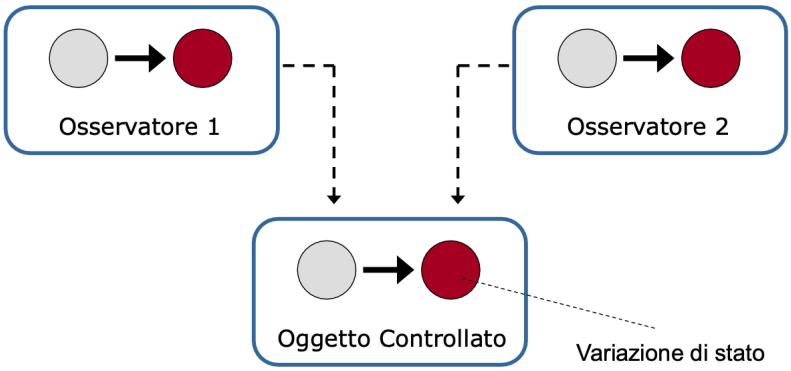
```
BufferedInputStream bin = new BufferedInputStream( new FileInputStream( "test.dat"));
```

(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).