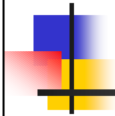


Object-Oriented Design Patterns

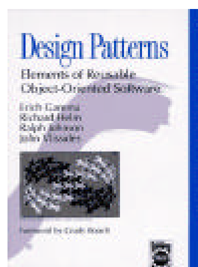
Riferimenti

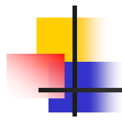
Design Patterns

Elements of Reusable Object-Oriented Software

by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

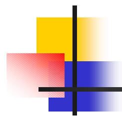
Addison-Wesley Publishing Company





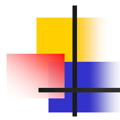
Introduzione

- La progettazione di software object-oriented è complessa, e lo è ancor di più la progettazione orientata al riuso
- Occorre identificare oggetti, fattorizzarli in classi con il giusto grado di granularità, definire le interfacce, le relazioni…
- Il progetto da un lato è specifico per il sistema che si va a realizzare, dall'altro dovrebbe essere sufficientemente generico in modo tale da venire incontro ad esigenze future.



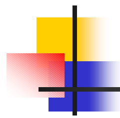
Non "re-inventare la ruota"

- Un buon progettista OO non cerca di risolvere daccapo ogni problema, bensì cerca di utilizzare soluzioni esistenti
- Di conseguenza, in un software OO possono essere presenti pattern di classi e oggetti comunicanti, atti a risolvere uno specifico problema
- I pattern rendono il progetto maggiormente flessibile, elegante e soprattutto **riusabile**



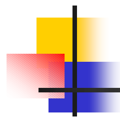
Cos'è un design pattern?

- Spesso, progettando un sistema, incontriamo problemi che abbiamo già risolto in precedenza (o che qualcun altro ha risolto)
- Una volta messi a fuoco i dettagli di un problema e risolto, è possibile riusare tale esperienza in futuro
- Un **design pattern** dunque identifica (con un nome), illustra e valuta ricorrenze all'interno di progetti OO, con l'obbiettivo di catturarle per permetterne un efficace riuso



Elementi di un pattern

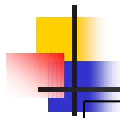
- **Nome:** identificatore utilizzato per descrivere un problema di design e la relativa soluzione
- **Problema:** quando applicare il pattern. Illustra il problema e il relativo contesto
- **Soluzione:** gli elementi che costituiscono il design e le relazioni tra essi
- **Conseguenze:** risultati derivanti dall'applicazione dei design pattern



Classificazione dei pattern

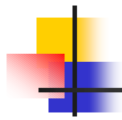
Classificheremo i pattern seguendo due criteri:

- Il primo, **purpose**, classifica i pattern in base a cosa il pattern faccia: i pattern possono dunque essere di tipo *creational, structural, behavioral*.
- Il secondo, **scope**, specifica se il pattern è applicabile a classi o a oggetti. Le relazioni insite nei pattern relativi agli oggetti sono dinamiche.



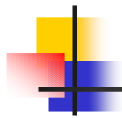
Classificazione dei pattern

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstact Factory Builder Prototype Sigleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsability Command Iterator Mediator Memento Observer State Strategy Visitor



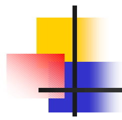
Selezione del pattern da usare

- Considerare come i pattern risolvono i problemi;
- Analizzare l'intento del pattern;
- Analizzare le relazioni tra diversi pattern;
- Confrontare tra loro i pattern di uno stesso purpose;
- Esaminare le ragioni del ridesign;
- Considera cosa potrebbe essere variabile nel design.



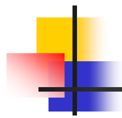
Come usare i pattern

- Porre molta attenzione sulle conseguenze;
- Studiare la *Struttura*, *Partecipanti* e *Collaborazioni*;
- Esaminare esempi di codice scritto usando i pattern;
- Scegliere, per i pattern utilizzati, nomi significativi nel contesto dell'applicazione da realizzare;
- Definire le classi partecipanti al pattern;
- Definire, per le operazioni, nomi significativi nel contesto dell'applicazione;
- Implementare le operazioni che realizzano le responsabilità e le collaborazioni del pattern.



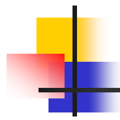
Creational patterns

- Un **creational pattern** aiuta a rendere un sistema indipendente da come gli oggetti sono creati, composti e rappresentati;
- Esistono due temi ricorrenti circa i creational patterns:
  - Incapsulare la conoscenza circa quale classe concreta il sistema utilizzi;
  - Nascondere il modo in cui istanze di classi siano create e messe assieme.
- In definitiva, si introduce flessibilità circa **cosa** è stato creato, **chi** lo ha creato, **come** lo ha creato, e **dove.**



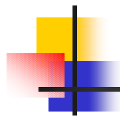
Creational patterns

- **Abstract Factory**
- **Builder**
- **Factory Method**
- **Prototype**
- **Singleton**

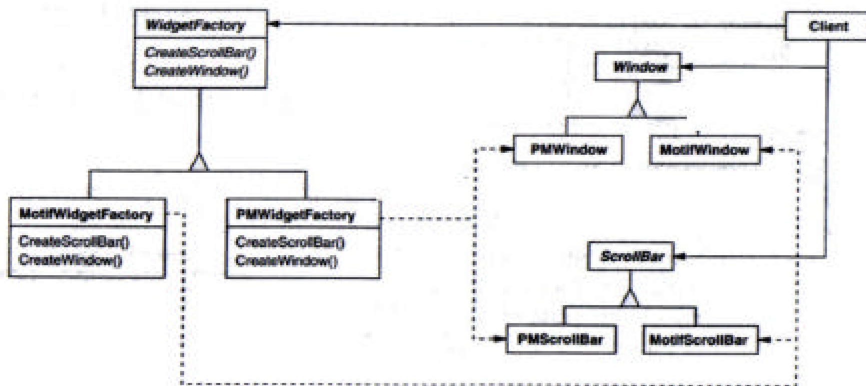


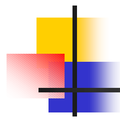
Abstract Factory (Object Creational)

- **Intento:** Fornire un'interfaccia per creare famiglie di oggetti dipendenti senza specificare le classi concrete;
- **Noto come:** kit
- **Motivazioni:** consideriamo es. Un interface toolkit che supporta differenti look-and-feel, che definiscono differenti comportamenti e notazioni grafiche per oggetti come bottoni, scrollbars, ecc. Un'applicazione portabile non dovrebbe legarsi ad uno specifico look-and-feel



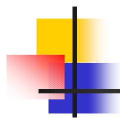
Abstract Factory (Object Creational)





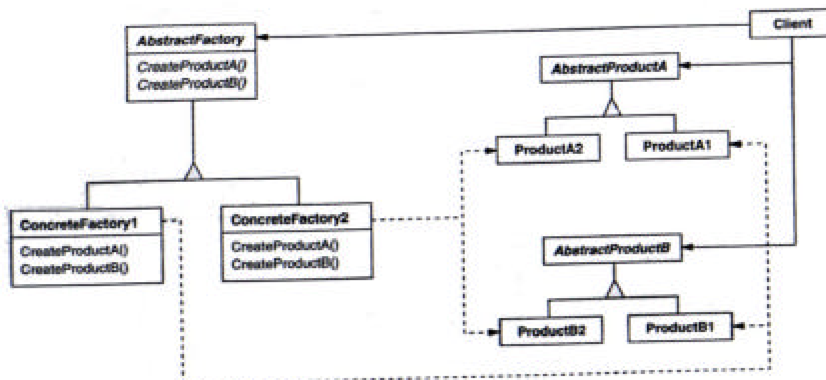
Abstract Factory (Object Creational)

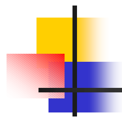
- **Applicabilità:**
  - Realizzare un sistema indipendente da come i prodotti sono creati, composti e rappresentati;
  - Il sistema deve essere configurato con famiglie multiple di prodotti;
  - Mettere a disposizione soltanto l'interfaccia, non l'implementazione, di una libreria di classi.



Abstract Factory (Object Creational)

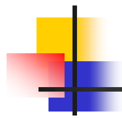
Struttura:





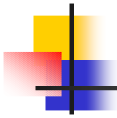
Abstract Factory (Object Creational)

- **Collaboration:**
  - Una singola istanza di ConcreteFactory è creata a run-time: tale istanza crea oggetti prodotto aventi particolari implementazioni
  - Per creare oggetti prodotto differenti, occorre usare differenti ConcreteFactory
  - La AbstractFactory delega la creazione di oggetti prodotto alla ConcreteFactory



Abstract Factory (Object Creational)

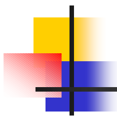
- **Conseguenze:**
  - Isola le classi concrete;
  - Facilita la portabilità;
  - Aumenta la consistenza tra i prodotti;
  - Per contro, inserire nuovi prodotti risulta complicato, in quanto implica cambiamenti all'*Abstract Factory*



Builder

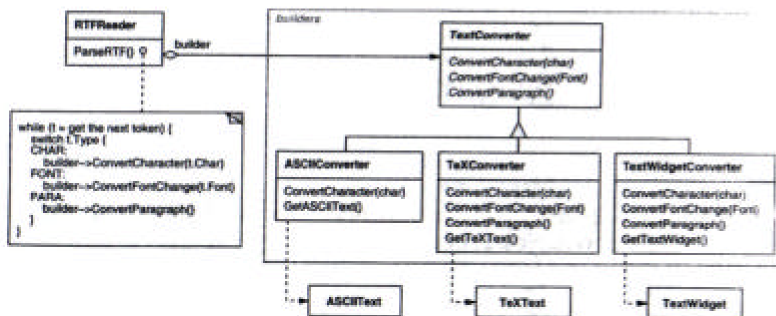
(Object Creational)

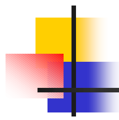
- **Intento:** separare la costruzione di un oggetto complesso dalla relativa rappresentazione
- **Motivazioni:** es. consideriamo uno strumento in grado di convertire documenti RTF in diversi formati. Occorrerebbe cercare di mantenere aperto il numero di possibili formati target.



Builder

(Object Creational)

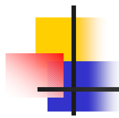




Builder

(Object Creational)

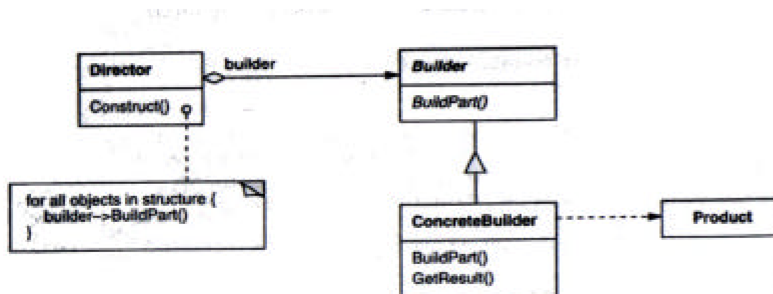
- **Applicabilità:**
  - L'algoritmo per la creazione di un oggetto complesso dovrebbe essere indipendente dalle componenti dell'oggetto stesso
  - Il processo di costruzione consente differenti rappresentazioni per l'oggetto

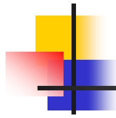


Builder

(Object Creational)

Struttura:

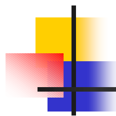




Builder

(Object Creational)

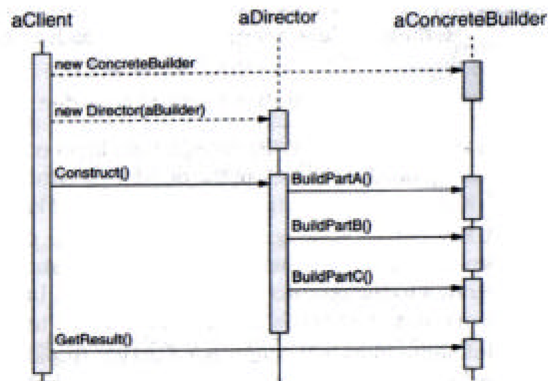
- **Collaborations:**
  - Il *client* crea il *Director* e lo configura col *Builder* desiderato;
  - Il *Director* notifica al *Builder* ogni qualvolta una parte del prodotto debba essere costruita;
  - Il *Builder* gestisce le richieste da parte del *Director* e aggiunge parti al prodotto;
  - Il *client* recupera il prodotto dal *Builder*.

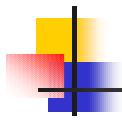


Builder

(Object Creational)

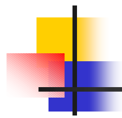
- **Collaborations:**





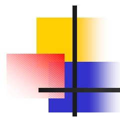
Builder (Object Creational)

- **Conseguenze:**
  - E' possibile variare la rappresentazione interna di un prodotto;
  - Isola il codice per la costruzione e la rappresentazione: il Builder incapsula il modo in cui un oggetto complesso è costruito;
  - Consente un miglior controllo sul processo di costruzione: il Builder consente una costruzione step by step del prodotto, sotto il controllo del Director.

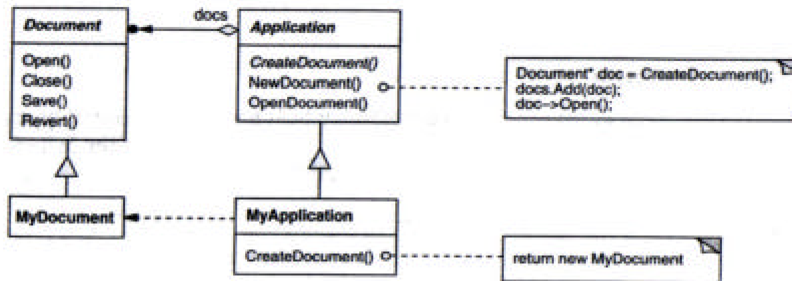


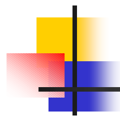
Factory Method (Class Creational)

- **Intento:** definire un'interfaccia per la creazione di un oggetto, delegando alle sottoclassi la decisione su quali classi istanziare.
- **Noto come:** Virtual Constructor
- **Motivazioni:** es. consideriamo un framework che presenta documenti multipli agli utenti. Le classi chiave sono *Application* e *Document*, entrambe astratte. Per creare un'applicazione per disegnare, occorre creare le classi *DrawingApplication* e *DrawingDocument.* Il pattern sposta all'esterno del framework la conoscenza di quale sottoclasse di documenti creare.



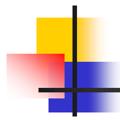
Factory Method (Class Creational)





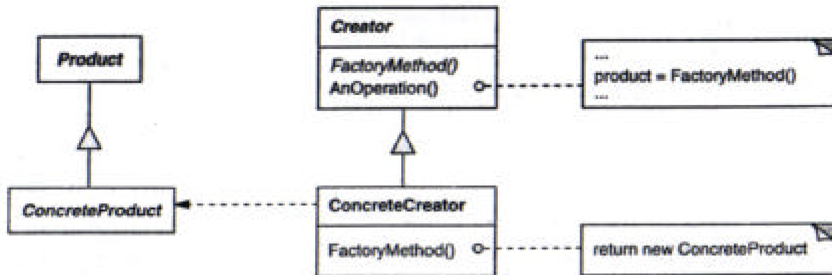
Factory Method (Class Creational)

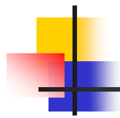
- **Applicabilità:** il pattern può essere utilizzato quando:
  - Una classe non può conoscere in anticipo la classe di oggetti che deve creare;
  - Una classe desidera che le proprie sottoclassi specifichino gli oggetti che creano;
  - Una classe delega responsabilità ad una delle proprie sottoclassi, e occorre localizzare la conoscenza della classe a cui delegare



Factory Method (Class Creational)

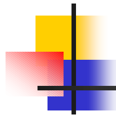
Struttura:





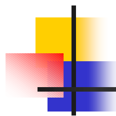
Factory Method (Class Creational)

- **Collaborations:**
  - Il *Creator* si affida alle proprie sottoclassi per definire il Factory Method, in maniera tale che questo ritorni un'appropriata istanza del *ConcreteProduct*
- **Conseguenze:**
  - Creare "agganci" alle sottoclassi, in maniera tale da fornire versioni estese di un oggetto;
  - Collegare gerarchie di classi parallele (es. L'operazione di "stretching" assume significati diversi su oggetti di tipo testo rispetto a oggetti di tipo disegno)

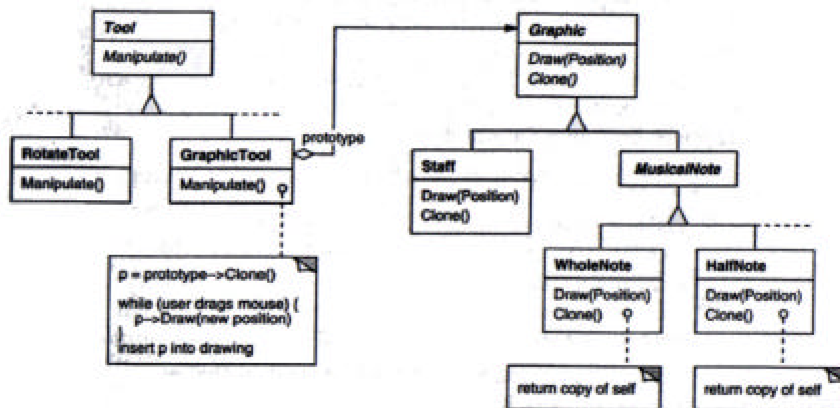


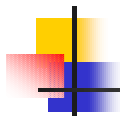
Prototype (Object Creational)

- **Intento:** Specificare il tipo di oggetti da creare utilizzando un'istanza prototipale, e creare nuovi oggetti copiando il prototipo.
- **Motivazioni:** Supponiamo di voler creare un editor per spartiti musicali a partire da un generico editor grafico: aggiungiamo nuovi oggetti che rappresentano note,… . Come è possibile notare, *GraphicTool* crea una nuova istanza di un *Graphic* copiando o "clonando" un'istanza di tale classe, chiamata **prototipo**. Il *GraphicTool* è parametrico rispetto al prototipo che clona e aggiunge al documento.



Prototype (Object Creational)

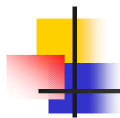




Prototype

(Object Creational)

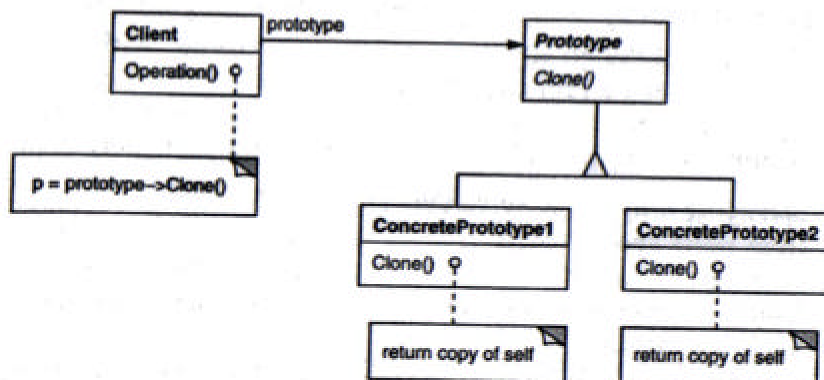
- **Applicabilità:** Il pattern andrebbe usato nel caso in cui un sistema dovrebbe essere indipendente da come i prodotti sono creati, e:
  - Quando le classi da istanziare sono specificate a run-time;
  - Per evitare la creazione di gerarchie di factory parallele alle gerarchie di prodotti;
  - Quando le istanze di una classe possono trovarsi in soltanto una (o poche) combinazioni di stati.

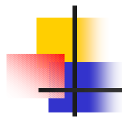


Prototype

(Object Creational)

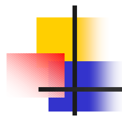
Struttura:





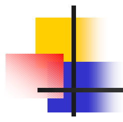
Prototype (Object Creational)

- **Collaborations:** un client chiede ad un prototipo di clonarsi
- **Conseguenze** (oltre a quelle dell'Abstract Factory e del Builder):
  - Aggiungere o rimuovere prodotti a run-time;
  - Specificare nuovi oggetti variando valori;
  - Specificare nuovi oggetti variando strutture;
  - Ridurre il subclassing;
  - Caricare classi dinamicamente nell'applicazione.



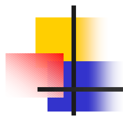
Singleton (Object Creational)

- **Intento:** Assicurarsi che una classe abbia soltanto un'istanza, e fornirne un unico punto di accesso
- **Motivazioni:** Es. nonostante in un sistema vi siano più stampanti, dovrebbe esserci soltanto un unico spool. Come assicurarsi che la classe spool abbia una sola istanza e che questa sia facilmente accessibile? Un modo è far si che la classe tenga traccia delle proprie istanze.



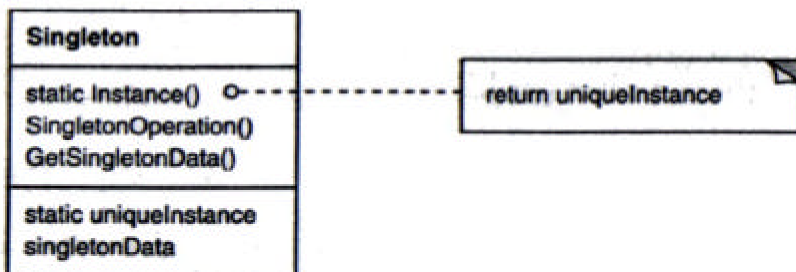
Singleton (Object Creational)

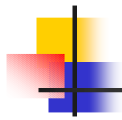
- **Applicabilità:** Il Singleton andrebbe usato se:
  - Deve esserci esattamente una singola istanza di una classe, e deve essere accessibile da un punto di accesso ben preciso;
  - Quando tale istanza deve essere estensibile tramite subclassing, i client dovrebbero poter utilizzare l'istanza estesa senza modificare il proprio codice.



Singleton (Object Creational)

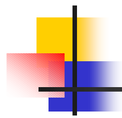
Struttura:





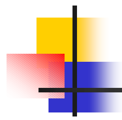
Singleton (Object Creational)

- **Collaborations:** I client accedono all'istanza del Singleton tramite l'operazione di *Instance* del Singleton stesso.
- **Conseguenze:**
  - Accesso controllato all'istanza singola;
  - Name space ridotto;
  - Raffinamento di operazioni e rappresentazione delle stesse;
  - Possibilità di usare un numero variabile di istanze;
  - Maggiore flessibilità rispetto all'uso degli static member.



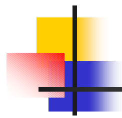
Structural patterns

- Riguardano il modo in cui classi e oggetti sono legati tra loro in strutture più grandi.
- I pattern strutturali utilizzano ereditarietà per comporre interfacce o implementazioni: es. è possibile definire un pattern che combina più classi. Tale pattern risulta utile per rendere interoperabili librerie sviluppate in maniera indipendente l'una dall'altra.
- Altro caso importante di pattern strutturale è l'Adapter, che rende un'interfaccia conforme ad un'altra differente.



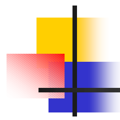
Structural patterns

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

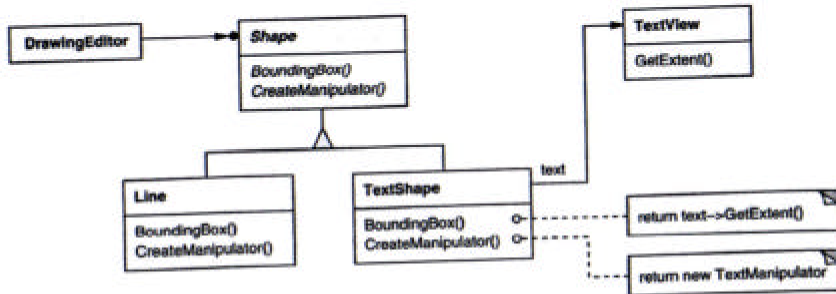


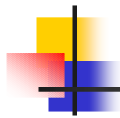
Adapter (Class, Object Structural)

- **Intento:** Convertire l'interfaccia di una classe in un'interfaccia differente richiesta dal client: in tal modo è possibile l'interoperabilità tra classi aventi interfacce incompatibili
- **Noto come:** Wrapper
- **Motivazioni:** Abbiamo a disposizione classi per figure geometriche (specializzazioni di *Shape*) e perr l'editing di testo (*TextView)*. Occorre realizzare un editor di testo/grafica *TextShape*, ma l'interfaccia di *TextView* non è conforme a quella di *Shape*



Adapter (Class, Object Structural)



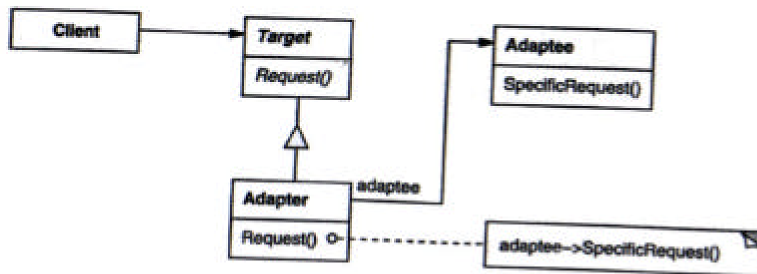


Adapter (Class, Object Structural)

- **Applicabilità:** è consigliabile utilizzare l'adapter quando:
  - Occorre utilizzare classi esistenti, ma le interfacce non sono compatibili con quella del client;
  - Occorre creare una classe riusabile che coopera con altre classi scollegate da essa, e dunque aventi interfacce diverse;
  - Occorre utilizzare diverse sottoclassi esistenti, ma non è pratico adattare l'interfaccia effettuando il subclassing di ognuna. Si può allora utilizzare un **object adapter**.

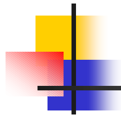
Adapter (Class, Object Structural)

Struttura:



Adapter (Class, Object Structural)

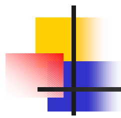
- **Collaborations:** i client invocano le operazioni tramite l'interfaccia dell'*Adapter*, che a sua volta richiama le operazioni dell'*Adaptee*
- **Conseguenze:**
  - Un *Class Adapter* non è in grado di adattare una classe e tutte le relative sottoclassi;
  - Un *Class Adapter* consente l'override del comportamento dell'Adaptee;
  - Un *Object Adapter* consente ad un singolo Adapter di operare con diversi Adaptee;
  - Tramite un *Object Adapter* l'overriding del comportamento dell'Adaptee risulta maggiormente difficoltoso: richiede il subclassing dell'Adaptee, e richiede che l'Adapter punti a tali subclassi piuttosto che all'Adaptee stesso.



Bridge

(Object Structural)

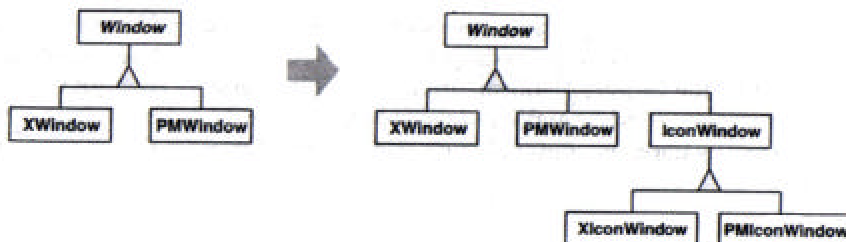
- **Intento:** Disaccoppiare un'astrazione dalla relativa implementazione in maniera tale da consentire ad entrambe di variare in maniera indipendente
- **Noto come:** Handle/body
- **Motivazioni:** Nel caso in cui un'astrazione possa avere diverse implementazioni, un modo di procedere è tramite l'ereditarietà. In questo caso, però, si lega l'implementazione all'astrazione in maniera permanente

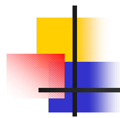


Bridge

(Object Structural)

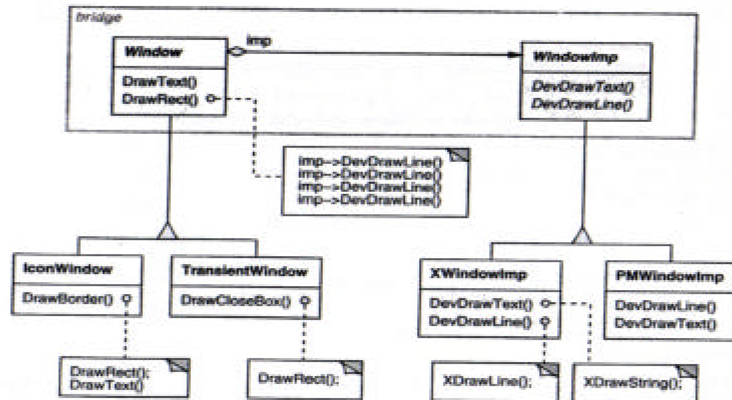
- **Esempio:** Supponiamo di avere la classe astratta *Window*, implementata per diversi sistemi, e specializzata per diversi tipi di finestre (es. *IconWindow*). Per ogni tipo di finestra dovremmo avere una specializzazione per ogni tipo di sistema.

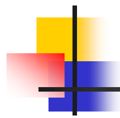




Bridge (Object Structural)

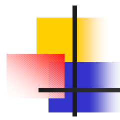
Il bridge separa la gerarchia dell'astrazione (*Window*) da quella dell'implementazione (*WindowImp*).





Bridge (Object Structural)

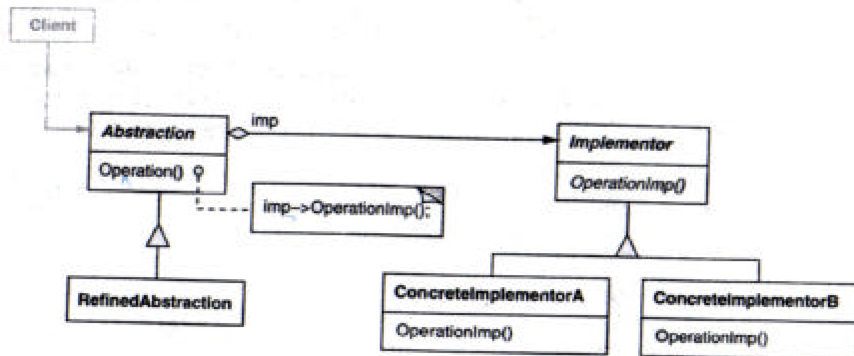
- **Applicabilità:** Può essere conveniente utilizzare il bridge nei seguenti casi:
  - Si desidera evitare un binding permanente tra astrazione e interfaccia;
  - Occorre rendere flessibili astrazione e implementazioni mediante subclassing;
  - Le modifiche all'implementazione non dovrebbero avere impatto sui client;
  - Rendere l'implementazione completamente invisibile (e quindi, in C++, anche l'interfaccia delle classi) ai clients;
  - Condividere un'implementazione tra oggetti multipli.

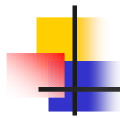


Bridge

(Object Structural)

Struttura

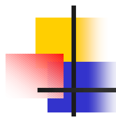




Bridge

(Object Structural)

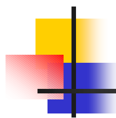
- **Collaborations:** Un'astrazione invia le richieste del client all'*Implementor*
- **Conseguenze:**
  - Disaccoppiare interfaccia e implementazione: ciò riduce anche necessità di ricompliazioni continue dell'Abstraction durante la fase di sviluppo, e incoraggia lo sviluppo di sistemi a layer
  - Migliorare l'estensibilità
  - Nascondere i dettagli implementativi ai client



Composite

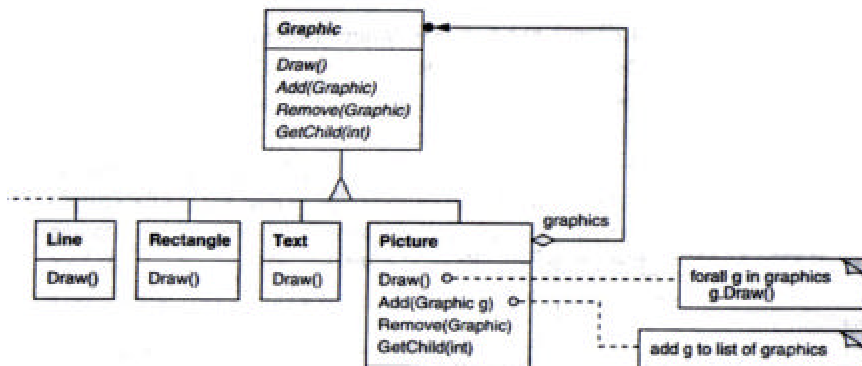
(Object Structural)

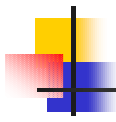
- **Intento:** Comporre oggetti in strutture ad albero per rappresentare gerarchie tutto-parti. Il composite consente di trattare oggetti singoli e composizioni di oggetti in maniera uniforme
- **Motivazioni:** Es. supponiamo di voler creare un programma di disegno, in grado di manipolare sia oggetti elementari come linee, testo, rettangoli, sia oggetti composti da oggetti elementari (figure).



Composite

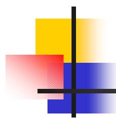
(Object Structural)





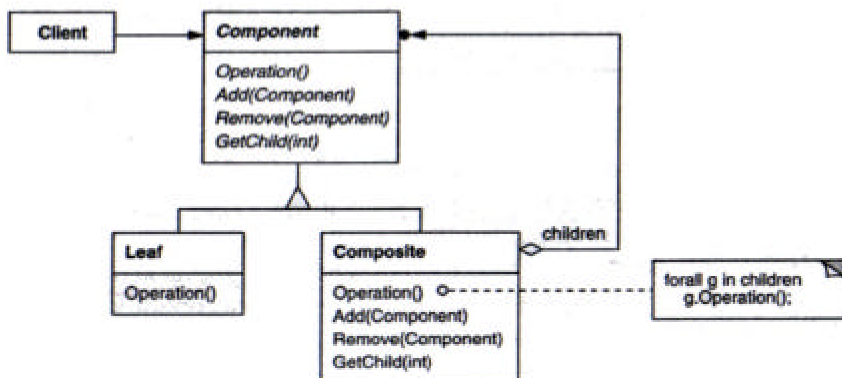
Composite (Object Structural)

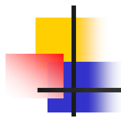
- **Applicabilità:** Il Composite dovrebbe essere usato quando:
  - **Si desidera rappresentare gerarchie tutto-parti**;
  - **Si cerca di rendere i client capaci di ignorare le differenze tra insiemi di oggetti ed oggetti individuali.**



Composite (Object Structural)

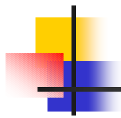
Struttura:





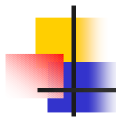
Composite (Object Structural)

- **Collaborations:** I client utilizzano l'interfaccia della classe *Component* per interagire con oggetti di strutture composte. Se il target è una foglia (*Leaf*) la richiesta è gestita direttamente, se è un *Composite*, questo inoltra la richiesta ai componenti figlio.



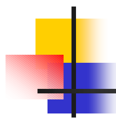
Composite (Object Structural)

- **Conseguenze:**
  - Sono definite gerarchie costituite da oggetti primitivi e oggetti composti;
  - Semplifica il client;
  - Rende semplice l'aggiunta di nuovi tipi di componenti;
  - Rende il design generale
  - E' però problematico porre dei vincoli sui componenti (es. specificare che un oggetto può essere composto solo da determinati componenti)

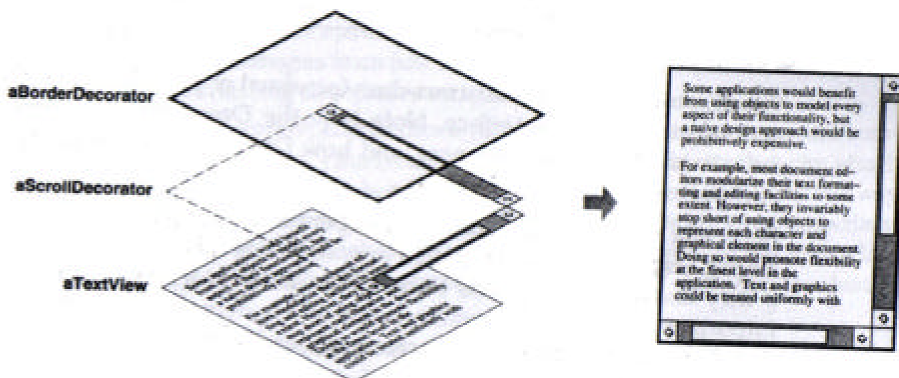


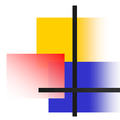
Decorator (Object Structural)

- **Intento:** Aggiungere responsabilità addizionali agli oggetti in maniera dinamica
- **Noto come:** Wrapper
- **Motivazioni:** Es. supponiamo in una GUI di avere un oggetto di tipo *TextView* e vogliamo aggiungervi responsabilità come bordi, scrollbar, ecc. Potremmo decidere di farlo tramite ereditarietà, ma in tal modo il meccanismo sarebbe statico (il client non può controllare quando e come aggiungere un bordo). Il *decorator* rende l'estensione trasparente all'interfaccia del componente che decora

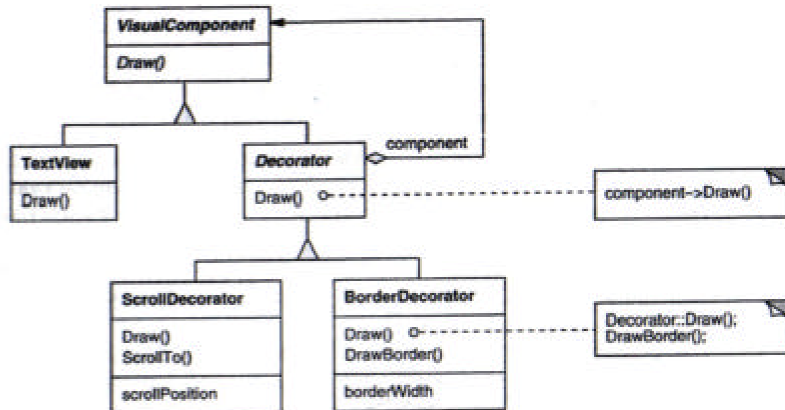


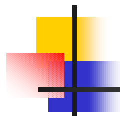
Decorator (Object Structural)





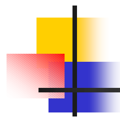
Decorator (Object Structural)





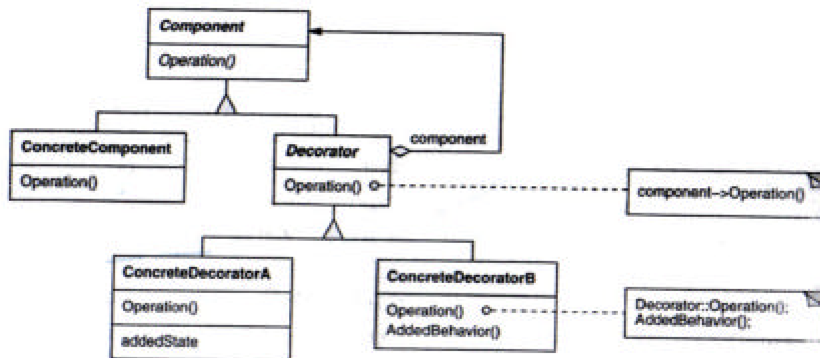
Decorator (Object Structural)

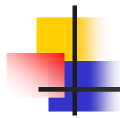
- **Applicabilità:** Il Decorator andrebbe usato:
  - **Per aggiungere responsabilità ad oggetti dinamicamente e trasparentemente;**
  - **Per aggiungere responsabilità che possono essere eliminate;**
  - **Quando l'estensione tramite subclassing si rivela poco pratica (es. un cospicuo numero di estensioni produrrebbe un'esplosione delle sottoclassi per supportare ogni combinazione).**



Decorator (Object Structural)

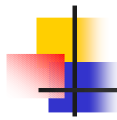
Struttura:





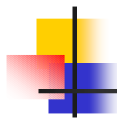
Decorator (Object Structural)

- **Collaborations:** Il Decorator inoltra le richieste agli oggetti Component e, opzionalmente, esegue altre operazioni
- **Conseguenze:**
  - Maggiore flessibilità rispetto all'ereditarietà statica;
  - Evita che classi relative a feature specifiche siano presenti nella parte alta della gerarchia;
  - Un Decorator e i suoi componenti non sono identici, anche se tutto ciò è trasparente dal punto di vista logico;
  - Usare i Decorator potrebbe però portare alla creazione di sistemi in cui sono presenti troppi oggetti molto piccoli, che differiscono tra loro solo per il modi in cui sono interconnessi.

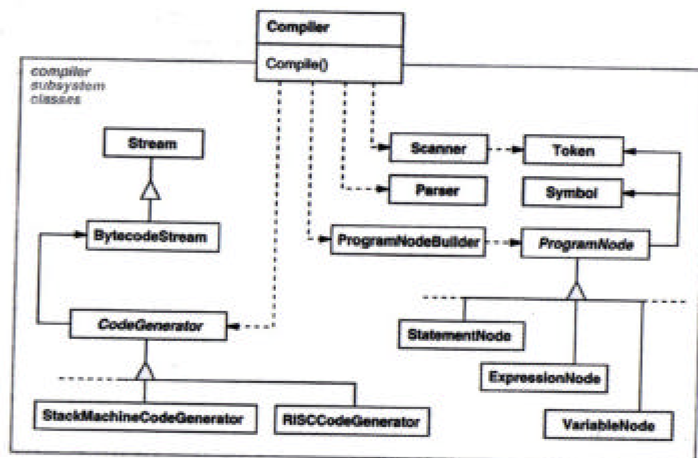


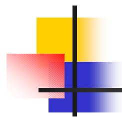
Facade (Object Structural)

- **Intento:** Fornire un'interfaccia unificata ad un insieme di interfacce di un sottosisistema, allo scopo di rendere il sottosistema più semplice da utilizzare.
- **Motivazioni:** Consideriamo es. un environment di programmazione, in cui è presente il sottosistema *Compiler*. Esso è costituito da classi come *Scanner*, *Parser*, *ProgramNode*, *ByteCodeStream*, ecc. In effetti, dall'esterno occorre soltato passare il sorgente e ricevere l'eseguibile/messaggi di errore. A questo punto introduciamo una classe *Compiler* che agisce da *Facade*.



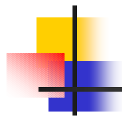
Facade (Object Structural)





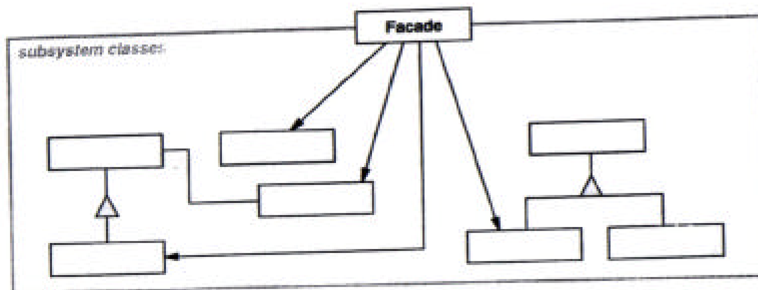
Facade (Object Structural)

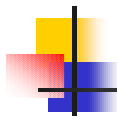
- **Applicabilità:** Il Facade andrebbe usato quando:
  - Si desidera fornire un'interfaccia semplificata ad un sottosistema complesso;
  - Esistono molte interdipendenze tra i client e le implementazioni delle astrazioni: il *facade* effettua un'operazione di disaccoppiamento;
  - Realizzare sistemi a layer, in cui il facade costituisce punto di accesso ad un layer (sottosistema).



Facade (Object Structural)

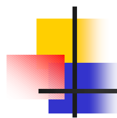
Struttura:





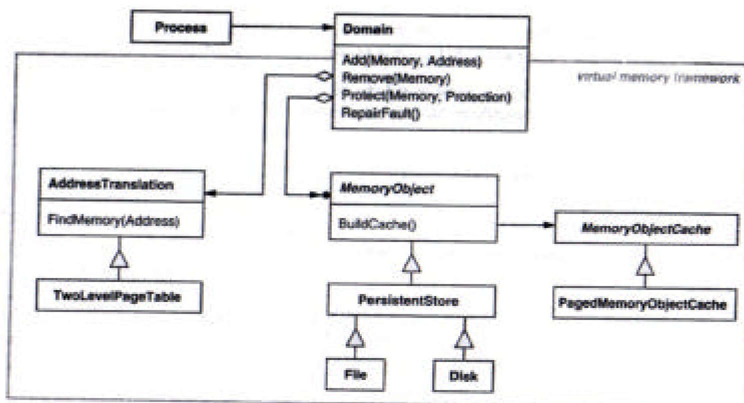
Facade (Object Structural)

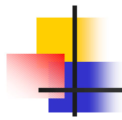
- **Collaborations:** I client comunicano col sottosistema inviando una richiesta al Facade, che si occupa di dispatcharla alle varie parti del sottosistema, traducendo opportunamente le richieste del client in maniera comprensibile agli oggetti del sottosistema;
- **Conseguenze:**
  - "Scherma" il client dai componenti del sottosistema, riducendo il numero di oggetti che il client manipola;
  - Favorisce un basso accoppiamento tra sottosistema e client;
  - In ogni caso, non proibisce ai client l'uso delle classi del sottosistema.



Facade (Object Structural)

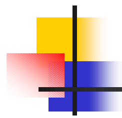
Esempio:





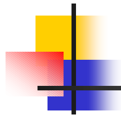
Flyweight (Object Structural)

- **Intento:** Utilizzare meccanismi di condivisione per utilizzare efficientemente oggetti a grana fine
- **Motivazioni:** Supponiamo che, in un editor di testi, si voglia rappresentare come oggetto ogni singolo carattere (in modo tale che ogni carattere abbia specifiche proprietà, come formato, colore, ecc.) e che l'applicazione tratti trasparentemente i vari caratteri. Ciò introduce overhead sia in termini di spazio che di prestazioni



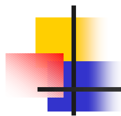
Flyweight (Object Structural)

- Un **flyweight** è un oggetto condiviso che può essere usato in contesti multipli simultaneamente, ed è indistinguibile rispetto ad un'istanza non condivisa di un oggetto.
- Il flyweight ha 2 stati: quello **intrinseco,** condiviso tra i diversi oggetti e contenuto all'interno del pattern, quello **estrinseco**, dipendente dal contensto, quindi non condiviso. Gli oggetti client sono responsabili del passaggio dello stato estrinseco al flyweight quando questi lo necessita.

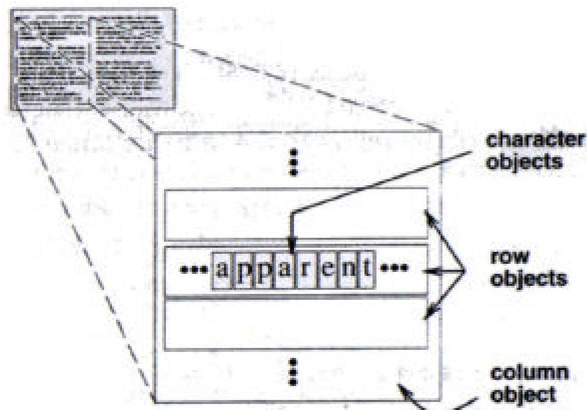


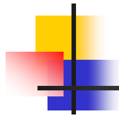
Flyweight (Object Structural)

- Nel caso dell'editor di testi, ciascun flyweight contiene il codice del carattere (stato *intrinseco*), ma la posizione nel documento e lo stile tipografico possono essere determinati all'occorrenza (stato *estrinseco*).
- Logicamente, vi è un oggetto per ogni carattere;
- Fisicamente, c'è un flyweight condiviso. Ciascuna occorrenza di uno stesso carattere punta alla medesima istanza nel pool di oggetti flyweight;
- Dunque, un flyweight che rappresenti la lettera "a" contiene solo il carattere, non la locazione o il font.



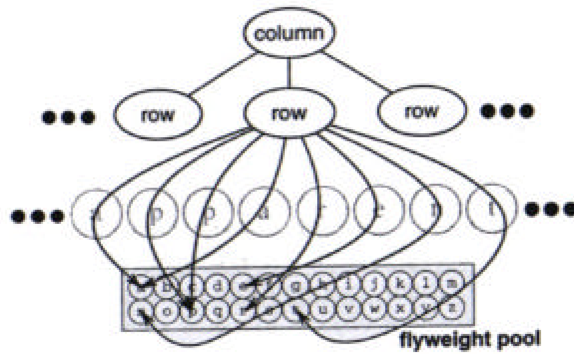
Flyweight (Object Structural)

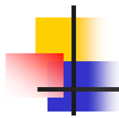




Flyweight

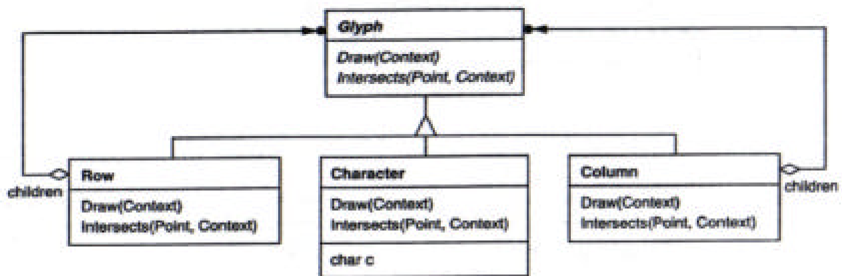
(Object Structural)

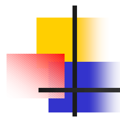




Flyweight

(Object Structural)

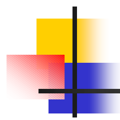




Flyweight

(Object Structural)

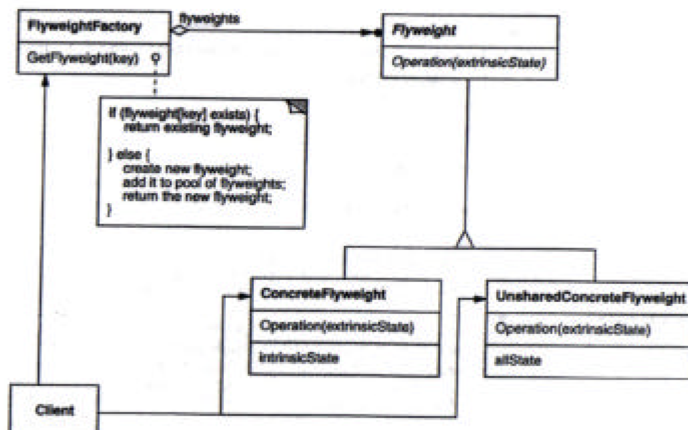
- **Applicabilità:** Il flyweight andrebbe usato soltanto se tutte le seguenti condizioni risultassero vere:
  - L'applicazione utilizza un largo numero di oggetti;
  - I costi di memorizzazione sono elevati a causa della grande quantità di oggetti;
  - La maggior parte dello stato dell'oggetto può essere resa estrinseca;
  - Diversi gruppi di oggetti possono essere sostituiti da relativamente pochi oggetti condivisi una volta rimosso lo stato estrinseco;
  - L'applicazione non è dipendente dall'identità dell'oggetto.

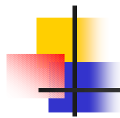


Flyweight

(Object Structural)

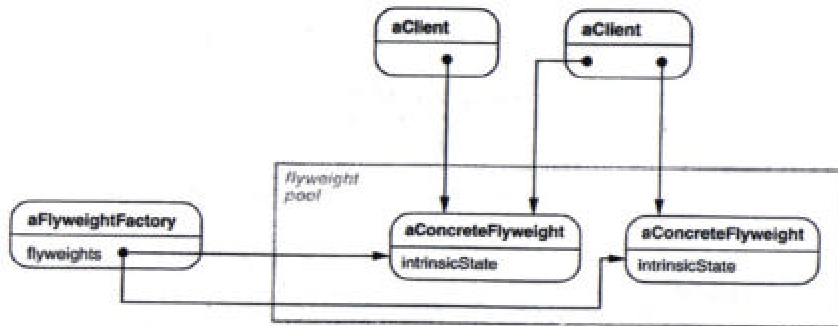
Struttura:

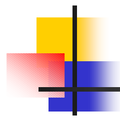




Flyweight (Object Structural)

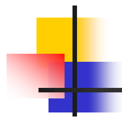
Meccanismo di condivisione:





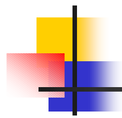
Flyweight (Object Structural)

- **Collaborations:** I client non dovrebbero istanziare un *ConcreteFlyweight* direttamente, ma ottenerli dalla *FlyweightFactory* in maniera tale che il meccanismo di condivisione funzioni correttamente.



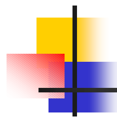
Flyweight (Object Structural)

- **Conseguenze:** I flyweight introducono costi a run-time associati alla ricerca, al trasferimento, e al calcolo dello stato estrinseco, specialmente se questo è formalmente immagazzinato nello stato instrinseco. Tali costi sono ripagati dal risparmio di spazio, funzione dei seguenti fattori:
  - Riduzione del numero di istanze in conseguenza della condivisione;
  - Dimensione dello stato intrinseco di ogni oggetto;
  - Possibilità di calcolare lo stato estrinseco piuttosto che immagazzinarlo.



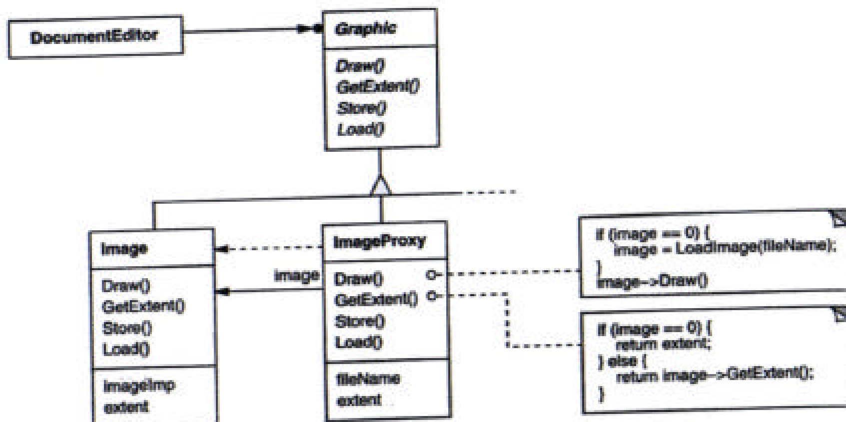
Proxy (Object Structural)

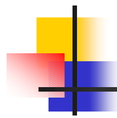
- **Intento:** fornire un "surrogato" di un altro oggetto in maniera di controllare l'accesso all'oggetto stesso.
- **Noto come:** Surrogate
- **Motivazioni:** Consideriamo un word processor in grado di includere immagini. Un'immagine di dimensioni considerevoli può essere costosa da creare. Ragion per cui, l'immagine viene creata *on demand* soltanto quando la pagina in cui è contenuta viene visualizzata. Un patter di tipo **proxy** si occupa di effettuare tale istanziazione alla richiesta dell'immagine.



Proxy

(Object Structural)

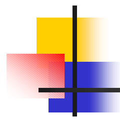




Proxy

(Object Structural)

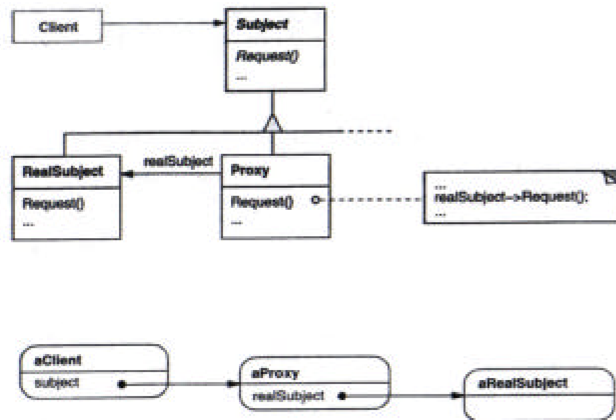
- **Applicabilità:** Un proxy è utile quando si necessità un riferimento ad un oggetto più versatile di un puntatore. Ecco alcuni esempi:
  - **Remote proxy:** fornisce una rappresentazione locale di un oggetto appartenente ad uno spazio di indirizzamento diverso;
  - **Virtual proxy:** crea oggetti "pesanti" on-demand. E' il caso del nostro esempio;
  - **Protection proxy:** Controlla l'accesso all'oggetto originale, il quale può avere differenti diritti di accesso.
  - **Smart reference:** Sostituzione di un semplice puntatore, che esegue operazioni aggiuntive quando avviene l'accesso all'oggetto.

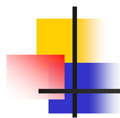


Proxy

(Object Structural)

Struttura:





Proxy

(Object Structural)

- **Collaborations:** il *Proxy* inoltra le richieste al *RealSubject* quando opportuno, in base al tipo di proxy realizzato
- **Conseguenze:**
  - Il *remote proxy* nasconde il fatto che un oggetto risieda in uno spazio di indirizzamento diverso;
  - Il *virual proxy* introduce ottimizzazioni creando oggetti on-demand;
  - Il *protection proxy* e la *smart reference* introducono livelli di protezione aggiuntivi.