Progettazione ad Oggetti (OOD) e Pattern di progetto

Corso di Ingegneria del Software

Anno Accademico 2012/2013

Progettazione ad oggetti (OOD)

- L'OOA identifica e definisce le classi e gli oggetti che riflettono il dominio del problema e le responsabilità del sistema entro di esso
- L'OOD identifica e definisce altre classi e oggetti, che riflettono un'implementazione dei requisiti
- Un'unica notazione (e la rappresentazione che sta sotto di essa) si applica sia all'analisi che alla progettazione.
- Nella fase di progettazione non è necessario creare un modello differente da quello sviluppato nell'OOA, poiché il paradigma ad oggetti serve ugualmente bene sia a modellare il mondo reale che all'implementazione del software
- Per passare all'OOD *occorre preliminarmente scegliere* l'architettura hw e sw, e l'architettura sw di base del sistema

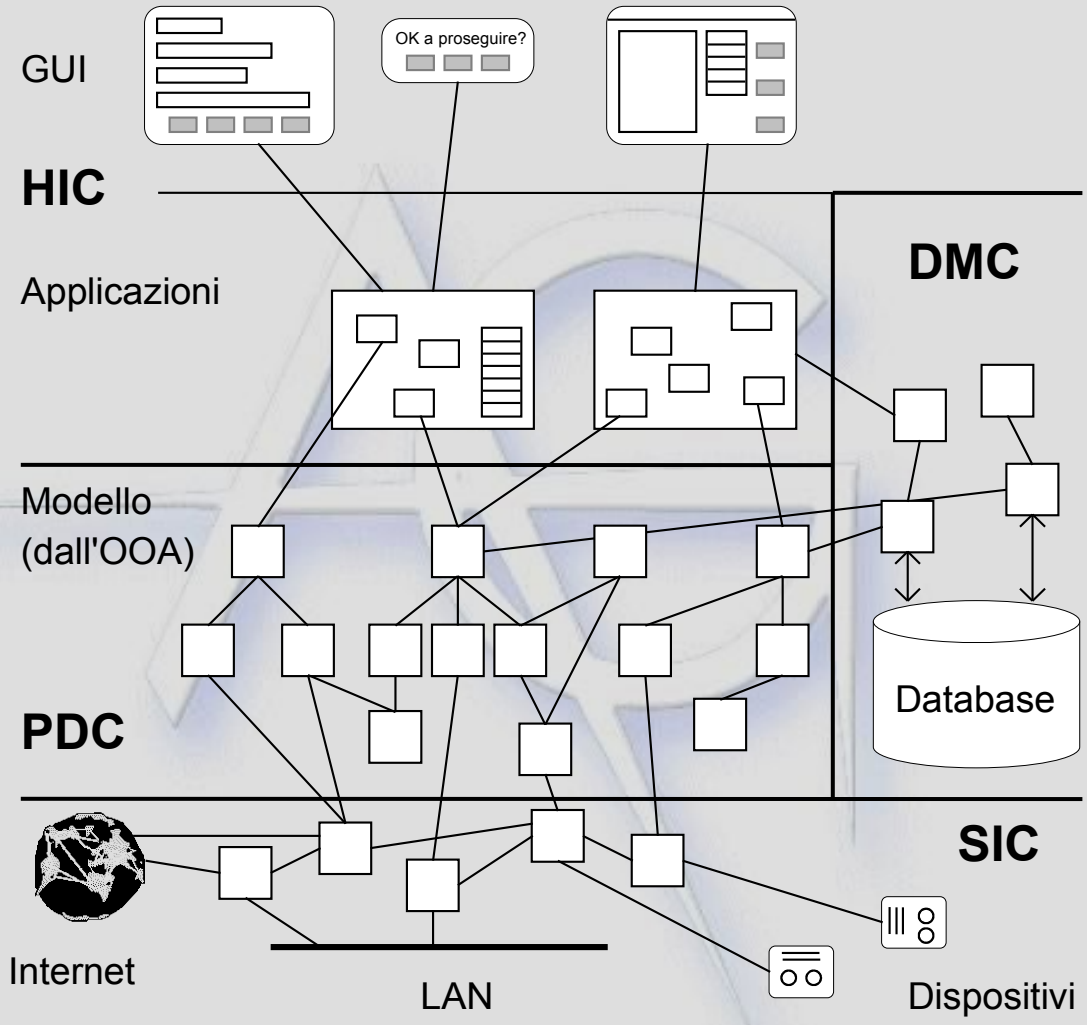
I cinque componenti dell'OOD

- Modello del sistema (PDC, deriva dall'OOA)
- Interazione Utente (HIC).
- Gestione dati permanenti (DMC).
- Interazione con sistemi esterni e reti (SIC)
- Gestione task (TMC)

Livelli degli oggetti del sistema

- **Modello**: dati intrinsici che descrivono il sistema.
- **Livello applicativo**: le applicazioni che manipolano i dati del modello interagendo con l'utente e/o con i database. Fanno parte del componente di Interazione Utente (HIC)
- **GUI**: finestre e parti che compongono l'interfaccia utente delle applicazioni. Fanno parte dell'HIC
- **Gestione dati**: oggetti preposti alla memorizzazione permanente degli oggetti del modello
- **Interazione con sistemi esterni e con la rete**

Livelli degli oggetti del sistema



Il Componente del Dominio del Problema (PDC)

- Nell'OOD, i risultati dell'OOA sono parte integrante del Componente del Dominio del Problema (PDC).
- I risultati dell'analisi si possono cambiare ed aumentare, per motivi di opportunità legati all'implementazione.
- Ovviamente, tali modifiche devono essere ridotte al minimo indispensabile, per mantenere il massimo accordo possibile tra OOA e OOD.

I motivi per modificare il modello dell'OOA

- Progettazione logica: definizione dettagliata dell'implementazione delle classi e delle loro relazioni.
- Riuso di classi disponibili.
- Livello di ereditarietà supportato dal linguaggio.
- Raggruppamento di classi del modello.
- Miglioramento delle prestazioni.
- Aggiunta di caratteristiche specifiche (gestione oggetti permanenti, gestione comunicazioni, diagnostica, ecc.).
- Supporto per l'interoperabilità su più sistemi.

“Pattern” di progettazione OO

- Un pattern è uno schema di oggetti per risolvere una determinata classe di problemi
- Gli oggetti di un pattern hanno specifiche responsabilità ed interazioni reciproche
- Lo schema si può applicare per analogia in molti progetti
- I pattern sono blocchi per costruire programmi, molto utili per ottenere modelli ad oggetti efficaci
- I pattern di progetto sono stati introdotti per primi nel libro "Design Patterns" di E. Gamma, R. Helm, R. Johnson, J. Vlissides, Addison-Wesley, 1995

Tipi di pattern

- pattern **creazionali**
  - Attengono alla creazione degli oggetti
- pattern **strutturali**
  - Attengono alla struttura dati degli oggetti
- pattern **comportamentali**
  - Attengono alle funzonalità degli oggetti

- Nel libro di Gamma et al. sono elencati 23 pattern.
- Altri pattern esistono in letteratura, per le più svariate applicazioni

Documentazione dei pattern

- **Nome:** identifica il pattern, e deve essere il più possibile descrittivo in una parola o due
- **Problema:** descrive quando usare il pattern, eventualmente con vincoli di applicabilità. Comprende **scopo** e **utilizzo**
- **Soluzione:** descrive gli elementi che costituiscono il pattern (**partecipanti**), la **struttura**, le **collaborazioni** tra oggetti, ed eventualmente il funzionamento
- **Conseguenze:** sono il risultato, e gli effetti collaterali dovuti all'applicazione del pattern. Esse sono fondamentali per valutare le alternative di progetto

I pattern creazionali

- forniscono schemi su come creare ed inizializzare nuovi oggetti
- Il loro scopo principale è il disaccoppiamento della creazione degli oggetti dalla specifica della loro classe, in modo da avere software più facilmente modificabile in caso di aggiunta di nuovi tipi di oggetti
- Forniscono schemi per evitare la chiamata diretta alla classe nel caso di creazione di un oggetto
- Aiutano ad avere software più modulare

Pattern: Costruttore (Builder)

- **Scopo**:

  Separa la costruzione di un oggetto complesso dalla sua rappresentazione, in modo che lo stesso processo possa creare differenti rappresentazioni

Utilizzo:

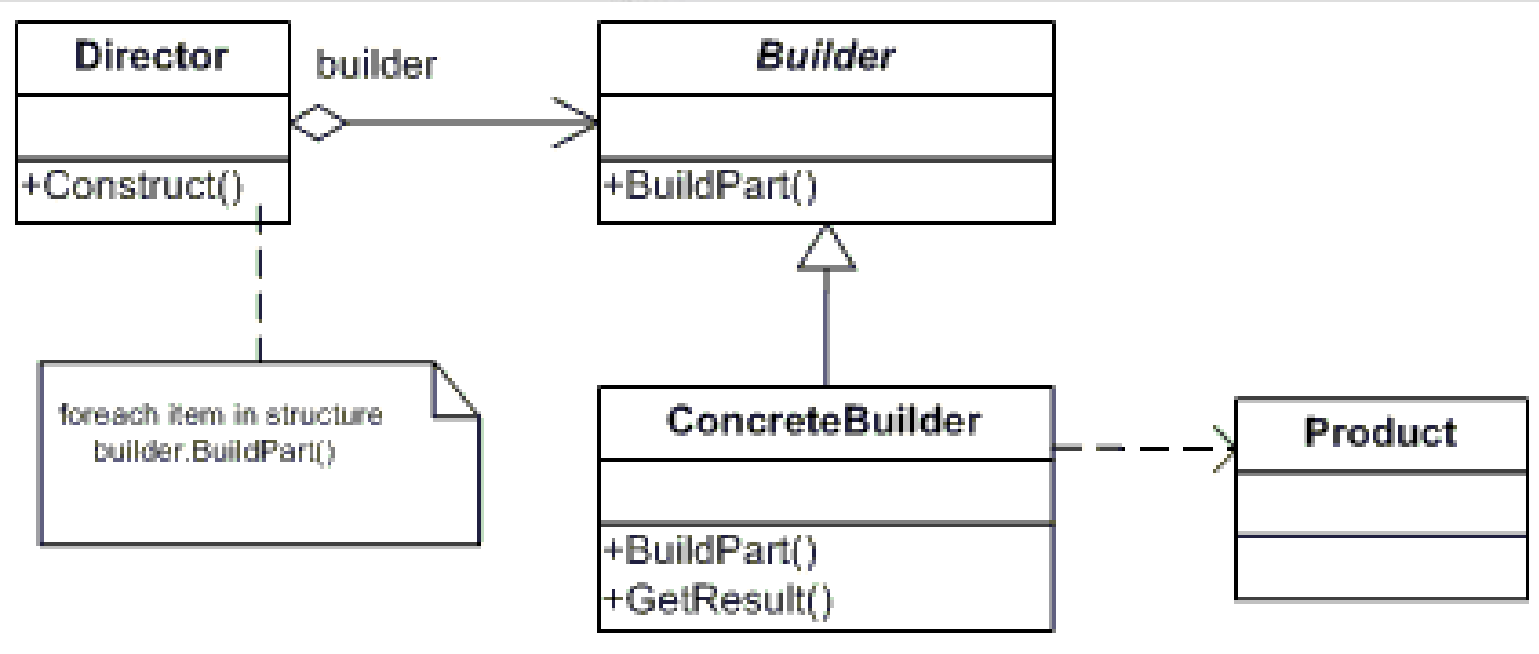
quando l'algoritmo per creare un oggetto complesso non deve dipendere dalle parti che lo compongono e da come sono messe insieme;

quando il processo di costruzione deve consentire diverse rappresentazioni dell'oggetto da costruire.

Partecipanti

- **Costruttore**:
  - specifica un'interfaccia astratta per creare le parti di un oggetto Prodotto
- **CostruttoreConcreto**:
  - costruisce e mette assieme le parti del prodotto, implementando l'interfaccia del Costruttore;
  - definisce e tiene traccia della rappresentazione da lui generata;
  - fornisce un interfaccia per accedere al prodotto.
- **Direttore**:
  - costruisce un oggetto usando l'intefaccia del Costruttore
- **Prodotto**:
  - rappresenta l'oggetto complesso in costruzione;
  - comprende le classi che definiscono le parti costituenti, con le interfacce per comporle insieme

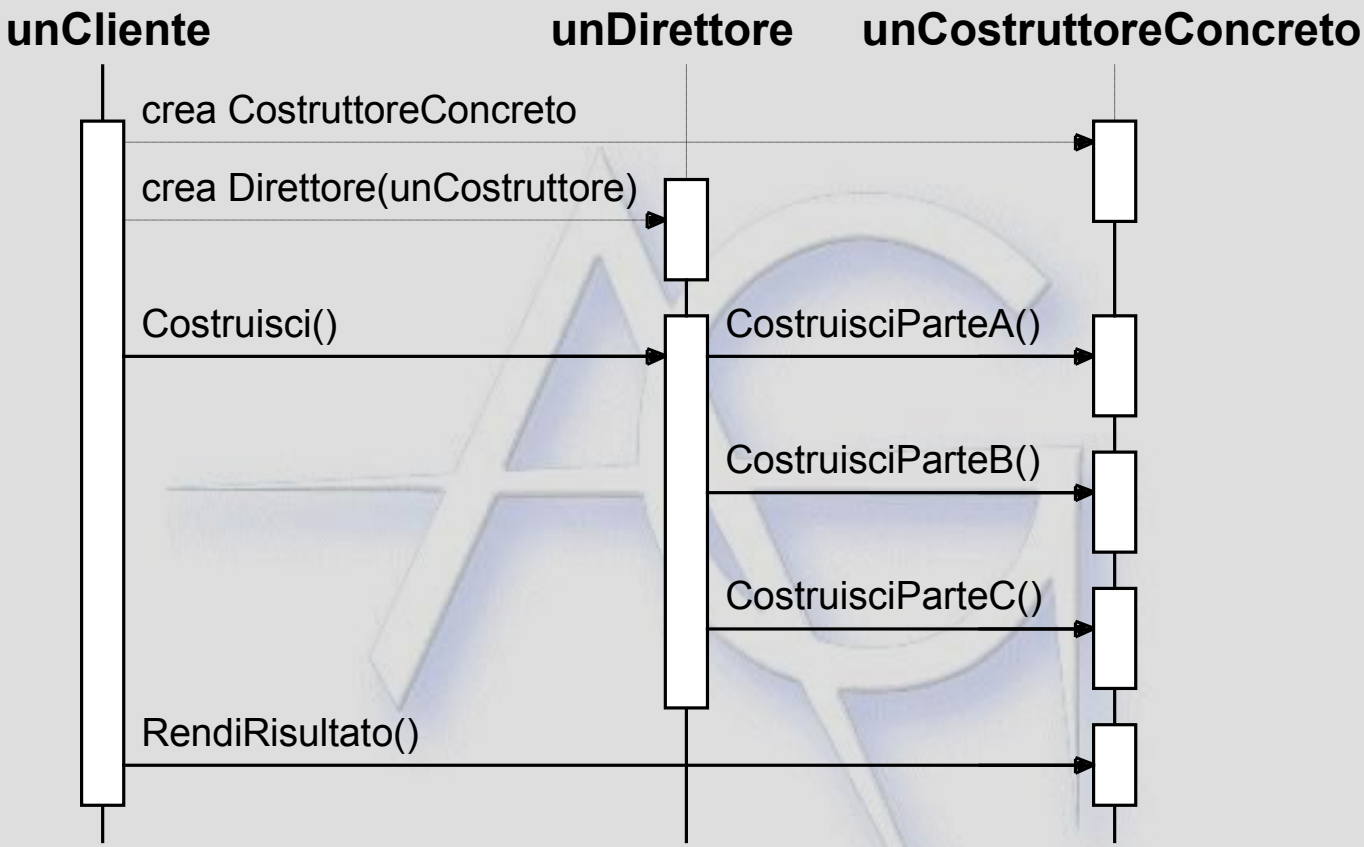
Struttura


Director
builder
Builder
+Construct()
+BuildPart()
foreach item in structure
builder.BuildPart()
ConcreteBuilder
+BuildPart()
+GetResult()
Product

Collaborazioni

- Il Cliente crea l'oggetto Direttore e lo configura col Costruttore voluto.
- Il Direttore comanda al Costruttore di creare le parti opportune.
- Il Costruttore esegue gli ordini del Direttore e aggiunge parti al Prodotto
- Il Cliente ottiene il Prodotto dal Costruttore

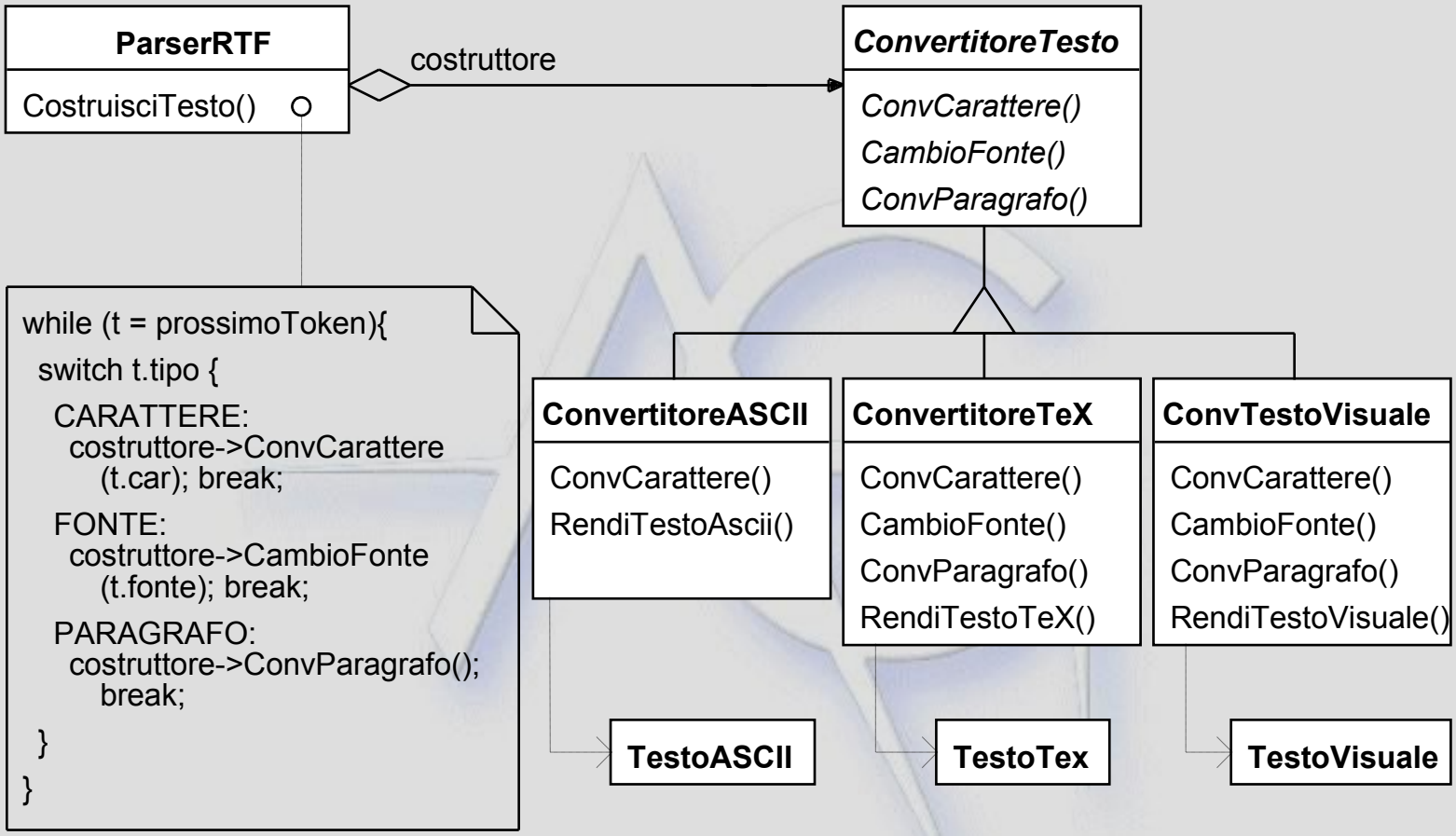
Diagramma sequenziale



Conseguenze

- La rappresentazione interna del prodotto può variare, senza che il Direttore o il Cliente ne siano influenzati.
- Isola il codice che costruisce il Prodotto, incapsulando la costruzione e la rappresentazione di un oggetto complesso
- Consente un maggior controllo del processo di costruzione, fatto passo-passo.

Esempio: conversione in txt, TeX o testo visuale



I pattern strutturali

- forniscono schemi su come comporre classi ed oggetti per formare strutture complesse
- I **pattern strutturali di classe** usano l'ereditarietà per comporre interfacce o implementazioni
- I **pattern strutturali di istanza** sono i più usati e descrivono modi per comporre oggetti, ottenendo nuove funzionalità.
- Gli scopi principali dei pattern strutturali sono il disaccoppiamento dell'accesso agli oggetti dalla loro specifica implementazione e la possibilità di aggiungere caratteristiche in modo incrementale
- Anche i pattern strutturali, quindi, forniscono schemi per avere software più modulare

Pattern: Composito (Composite)

- **Scopo**:

  Rappresentare in modo uniforme oggetti che contengono altri oggetti, sia semplici che a loro volta composti.

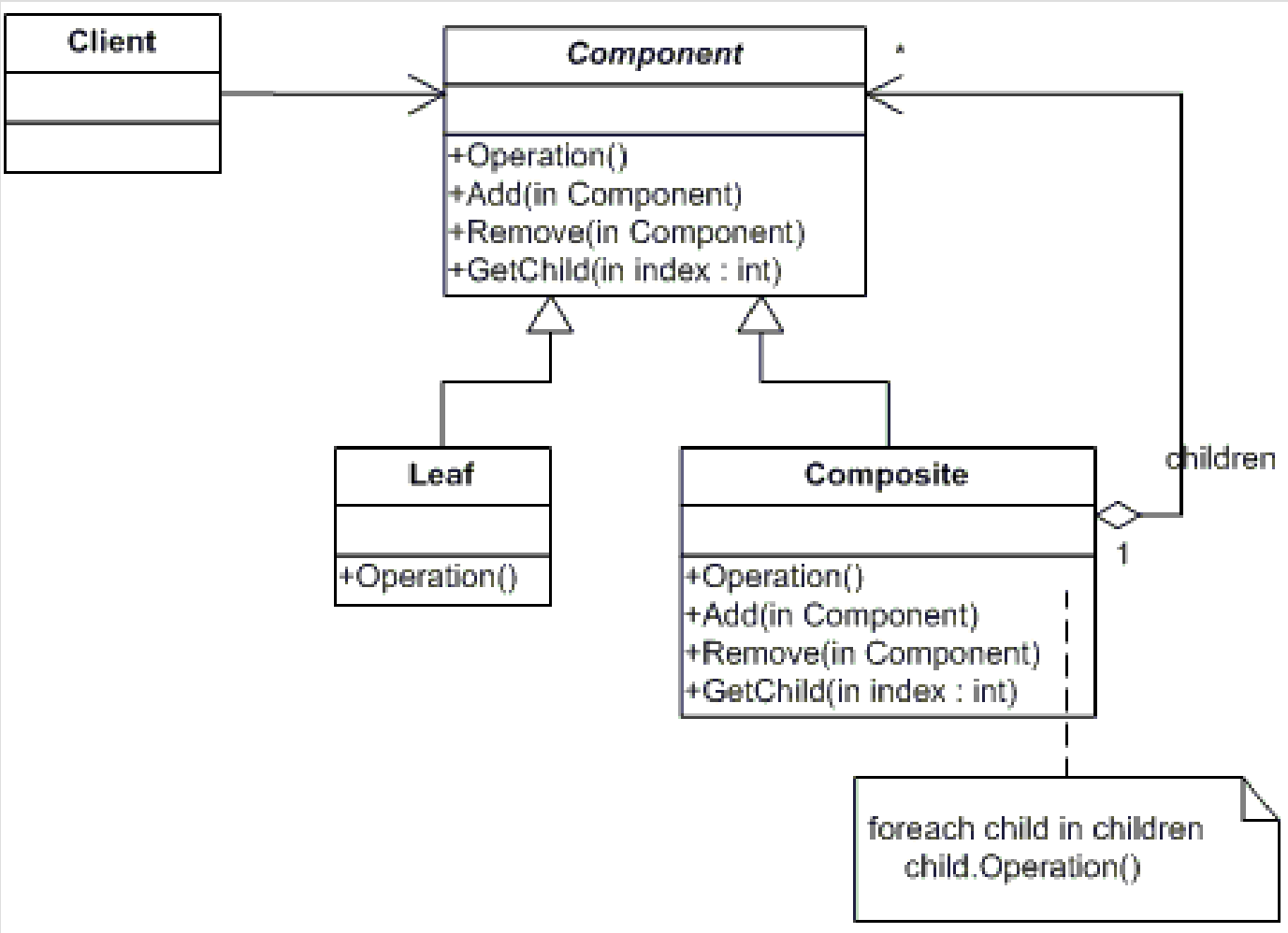
- **Utilizzo:**

  quando si vogliono rappresentare oggetti che contengono gerarchie di altri oggetti, in modo che l'interfaccia degli oggetti semplici (foglie) e di quelli composti sia uniforme.

Partecipanti

- **Componente**:
  - definisce l'interfaccia degli oggetti di una composizione;
  - implementa il comportamento di default di tutte le loro classi, se esiste;
  - definisce l'interfaccia per gestire i componenti "figli" (quelli contenuti in un "padre").
- **Foglia**:
  - rappresenta gli oggetti foglia, quelli che non contengono altri oggetti;
  - definisce il comportamento degli oggetti primitivi della composizione.
- **Composito**:
  - definisce il comportamento degli oggetti che ne contengono altri;
  - contiene oggetti "figli" ed implementa la gestione di tali figli.
- **Cliente**:
  - manipola gli oggetti della composizione usando l'interfaccia di Componente.

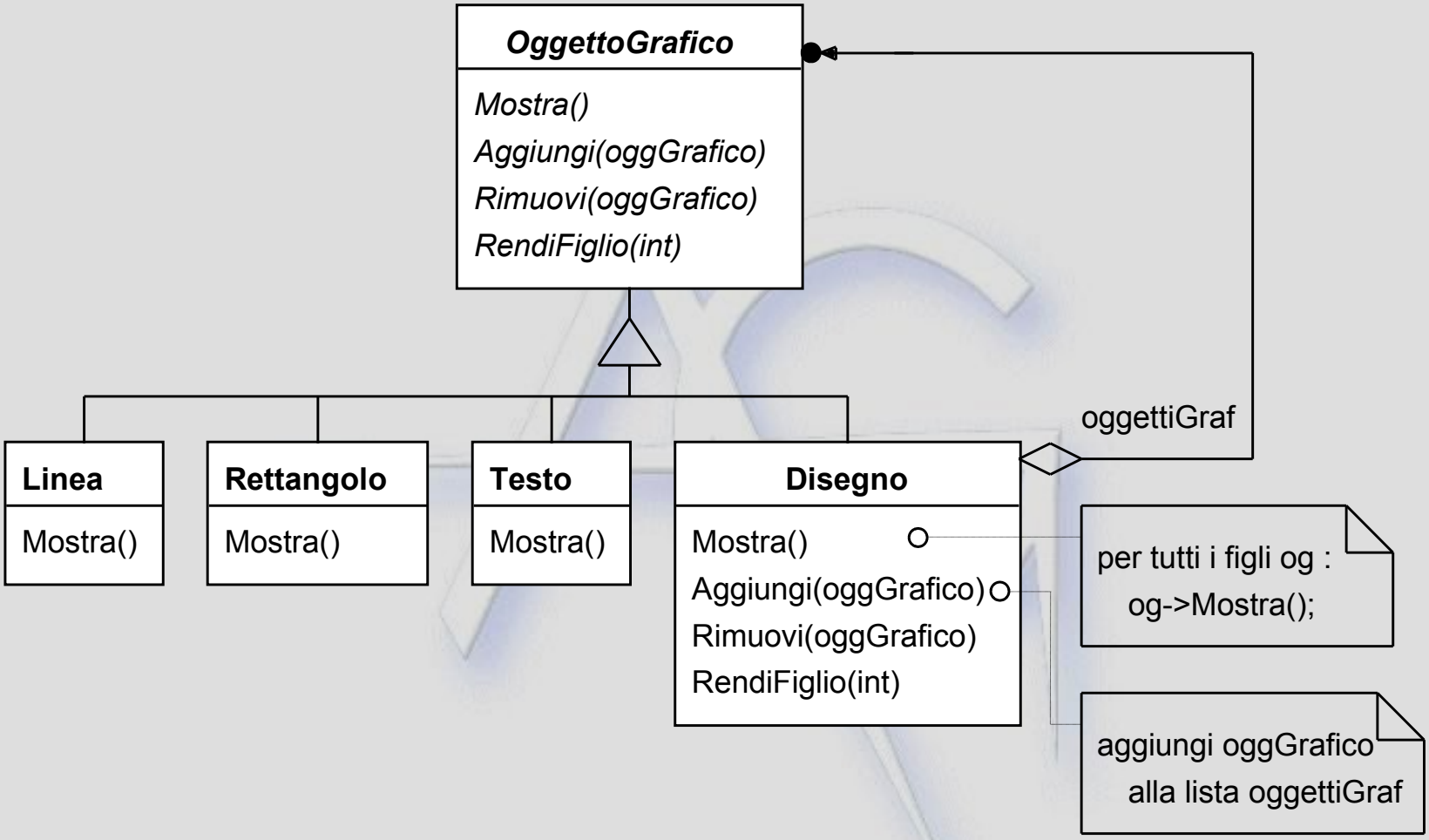
Struttura


Client
Component
+Operation()
+Add(in Component)
+Remove(in Component)
+GetChild(in index : int)
*
Leaf
+Operation()
Composite
+Operation()
+Add(in Component)
+Remove(in Component)
+GetChild(in index : int)
children
1
foreach child in children
child.Operation()

Collaborazioni

- I Clienti usano l'interfaccia della classe Componente per interagire con gli oggetti della struttura complessa. Se il ricevente è una Foglia, esegue direttamente la richiesta; se è un Composito, la passa ai suoi Componenti, eventualmente eseguendo operazioni aggiuntive
- **Conseguenze**:
  - Un Composito permette di definire gerarchie di classi composte da oggetti primitivi che si possono raggruppare in oggetti complessi, a vari livelli.
  - I Clienti possono trattare oggetti composti e oggetti primitivi in modo uniforme, e sono quindi liberati dalla gestione esplicita di tali strutture.
  - È più facile aggiungere nuovi oggetti, Foglia o Compositi.

Esempio: editor grafico



I pattern comportamentali

- Forniscono schemi che riguardano gli algoritmi e la suddivisione di responsabilità tra oggetti
- Anche i pattern comportamentali possono essere di classe o d'istanza. Questi ultimi sono i più comuni.
- I **pattern comportamentali di istanza** descrivono modi che usano la composizione di oggetti in modo da farli cooperare per ottenere un comportamento complesso
- Tendono ad avere la massima distribuzione delle responsabilità e la massima uniformità nelle interfacce.
- Il loro scopo principale è il disaccoppiamento degli oggetti cooperanti, in modo da poter riusare più facilmente i singoli oggetti e da facilitare le modifiche del software.

Pattern: Osservatore (Observer)

- **Scopo**:

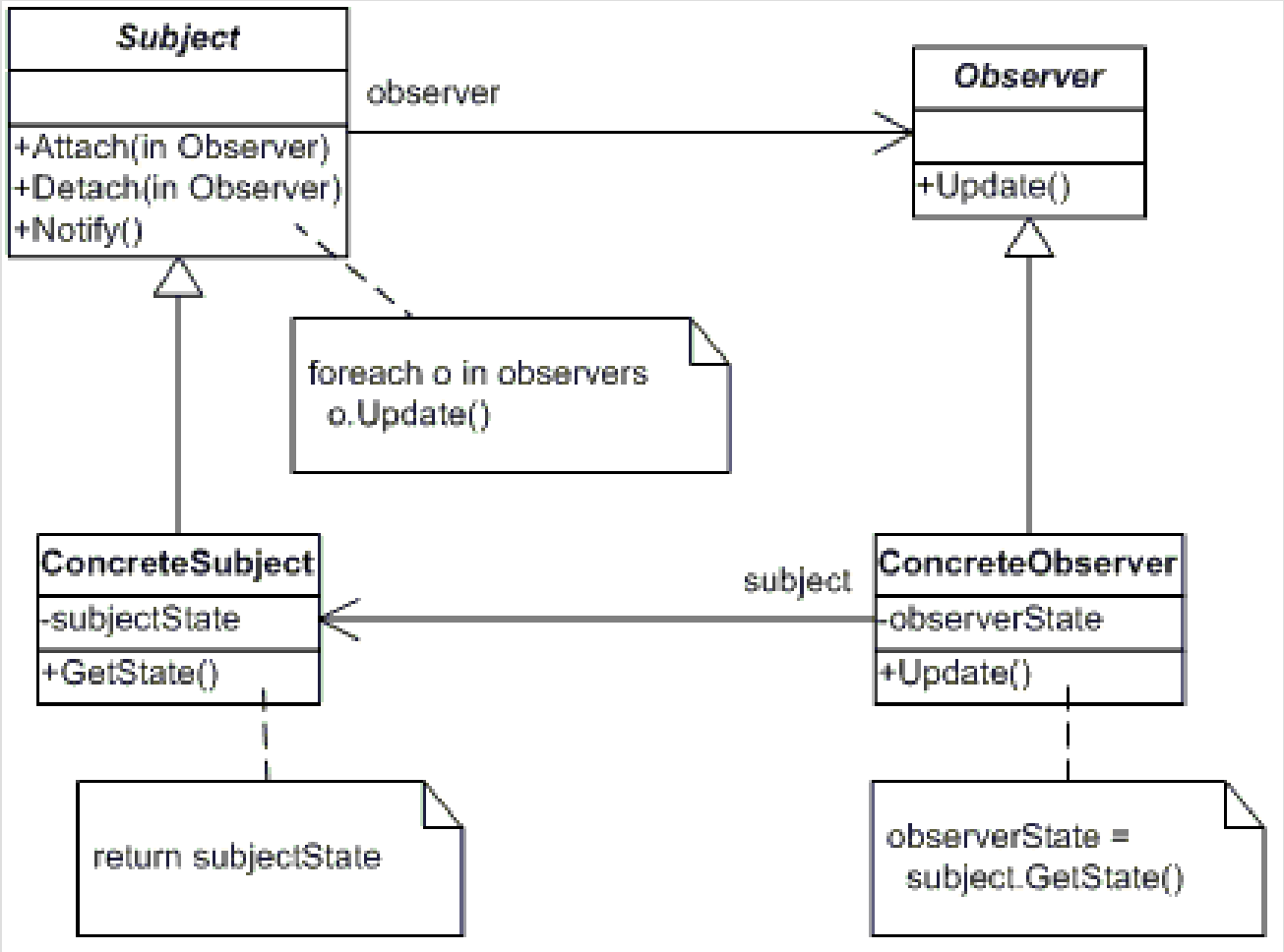
  Definire una dipendenza da uno a molti tra oggetti in modo che quando un oggetto cambia stato tutti i suoi dipendenti ne siano automaticamente notificati.
- **Sinonimi:**

  Dipendenti, Publish-Subscribe.
- **Utilizzo:**
  - quando un'astrazione ha due aspetti indipendenti, da incapsulare in oggetti separati da poter variare e riusare separatamente;
  - quando la modifica di un oggetto implica modifiche ad altri oggetti non noti a priori;
  - se un oggetto deve notificare informazioni ad altri oggetti senza doverli conoscere direttamente, mantenendo quindi un accoppiamento debole.

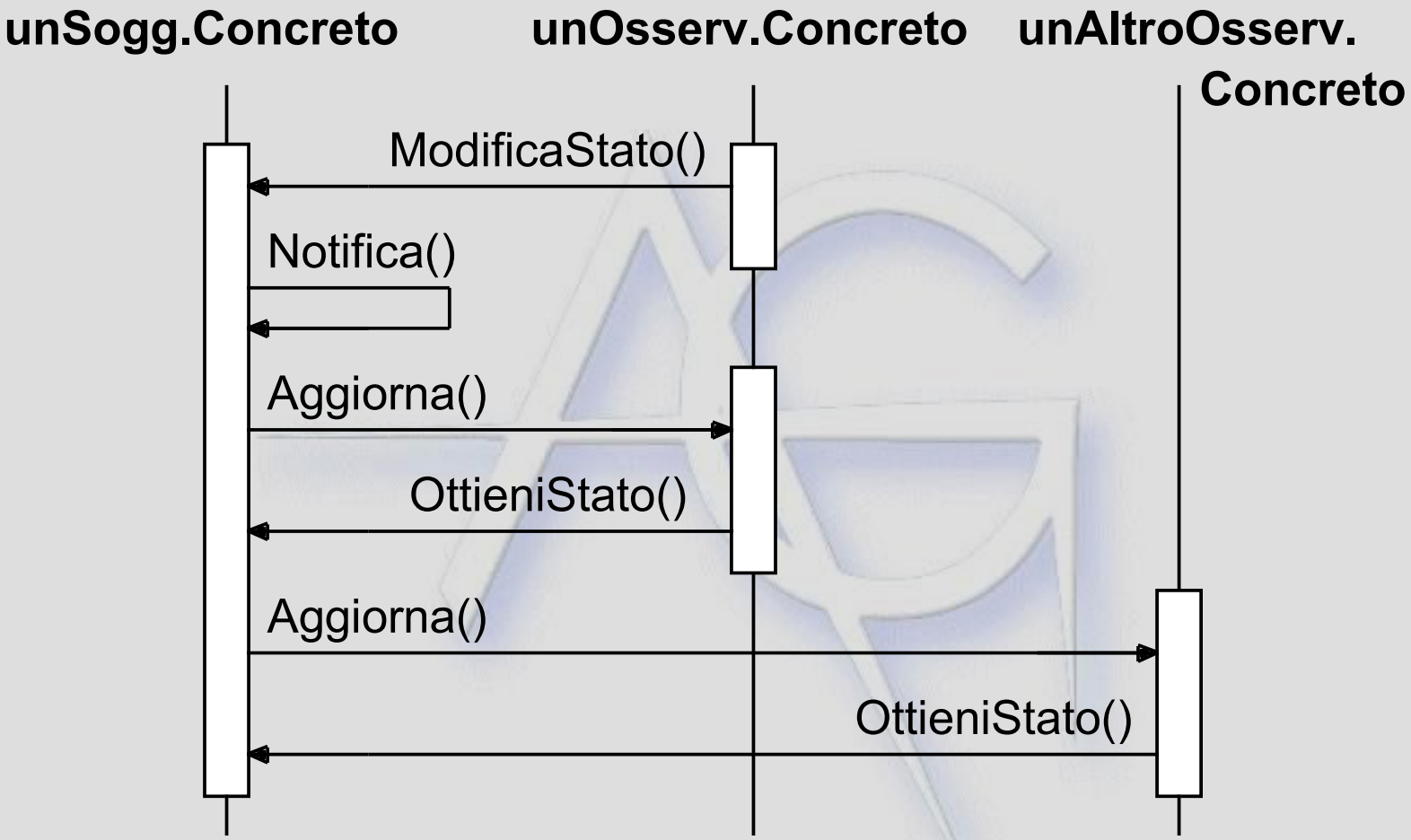
Partecipanti

- **Soggetto**:
  - conosce i propri Osservatori, che possono essere in numero qualsiasi;
  - fornisce un'interfaccia per aggiungere e togliere Osservatori da se stesso;
- **Osservatore**:
  - definisce un'interfaccia per aggiornare gli oggetti cui vanno notificati i cambiamenti.
- **SoggettoConcreto**:
  - memorizza i dati che interessano i propri Osservatori Concreti;
  - notifica i propri Osservatori Concreti quando cambia stato.
- **OsservatoreConcreto**:
  - ha un riferimento ad un Soggetto Concreto;
  - ha uno stato che deve essere coerente con quello del proprio Soggetto Concreto;
  - implementa l'interfaccia dell'Osservatore per mantenere il proprio stato coerente con quello del proprio Soggetto Concreto.

Struttura


Subject
observer
Observer
+Attach(in Observer)
+Detach(in Observer)
+Notify()
+Update()
foreach o in observers
o.Update()
ConcreteSubject
-subjectState
+GetState()
subject
ConcreteObserver
-observerState
+Update()
return subjectState
observerState =
subject.GetState()

Interazioni



Collaborazioni e conseguenze

- **Collaborazioni**:
  - Un Soggetto Concreto notifica i propri osservatori quando avviene un cambiamento nel proprio stato.
  - Dopo essere stato notificato del cambiamento, un Osservatore Concreto può richiedere informazioni al proprio soggetto per riconciliare il proprio stato con esso.
- **Conseguenze**:
  - L'accoppiamento tra Soggetto ed Osservatore è astratto: il Soggetto sa solo che ha una lista di Osservatori, e che deve notificare loro quando cambia. La classe specifica degli Osservatori e del Soggetto non ha importanza.
  - La notifica di cambiamento non deve specificare il ricevente, ma è inviata automaticamente a tutti gli Osservatori attaccati al Soggetto.