Design Patterns

fonti: [Gamma95] e [Pianciamore03]

Autori: Giacomo Gabrielli, Manuel Comparetti

Definizione

*"Ogni **pattern** descrive un problema che si presenta frequentemente nel nostro ambiente, e quindi descrive il nucleo della soluzione così che si possa impiegare tale soluzione milioni di volte, senza peraltro produrre due volte la stessa realizzazione."*

C. Alexander, architetto (di edifici, non di software)

- La definizione è valida anche per l'Ingegneria del SW, solo che gli elementi sono oggetti, classi, interfacce...

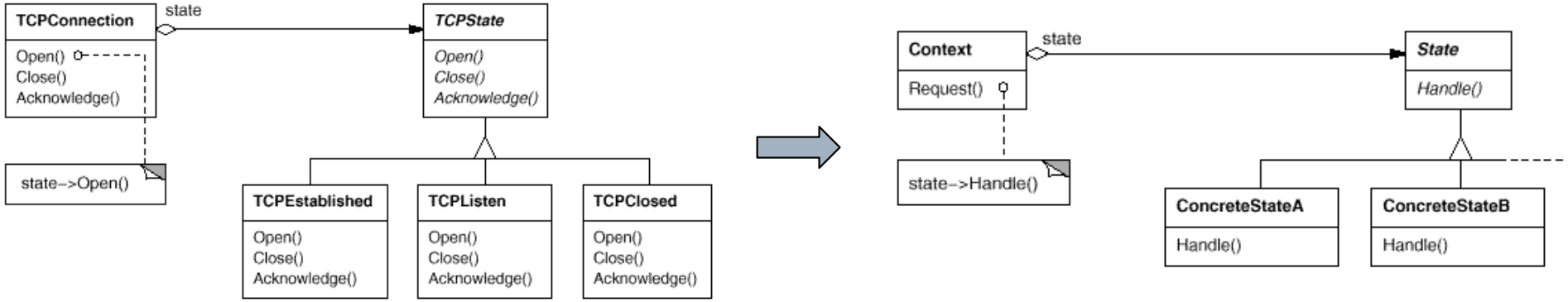
… “problemi”

- **Generici**: mappare una soluzione software in una gerarchia di classi. Progettare correttamente l’interfaccia degli oggetti. Ottimizzare l’interazione tra oggetti e minimizzarne le dipendenze. Produrre codice facilmente “estendibile”.
- **Specifici**: …come semplifico per il resto del sistema la navigazione di una struttura dati complessa? come separo interfaccia da implementazione in modo da poterle svilupparle indipendentemente? come aggiungo un nuovo algoritmo ad un insieme di classi che lo devono utilizzare senza doverle riscrivere tutte? come posso realizzare questo dinamicamente senza introdurre complicate strutture di controllo?…
- Un linguaggio di programmazione come il C++ ci offre i meccanismi per farlo, ma utilizzarli correttamente è compito nostro.

Definizione (pratica)

"Descrizioni di oggetti e classi in comunicazione tra loro che vengono personalizzati per risolvere un problema generale di progetto in un contesto particolare."

Es.: *State*



caso particolare

generalizzazione

Definizione (ii)

4 elementi essenziali:

- **Nome del pattern**
  espressione per descrivere univocamente un problema di progetto, le sue soluzioni e le sue conseguenze in una o due parole
- **Problema**
  descrive quando applicare il pattern (problema e contesto)
- **Soluzione**
  descrive gli elementi che compongono il design, le loro relazioni, responsabilità e collaborazioni
- **Conseguenze**
  risultati e compromessi derivanti dall'applicazione del pattern

Perché è importante studiarli

- **A nessun programmatore verrebbe in mente di ri-implementare una libreria che funziona, o scrivere ogni volta da capo una lista-> questo vale anche per i design pattern nel campo della progettazione software, ma l'approccio è diverso dall'utilizzo di librerie**
- **"Don't reinvent the wheel"**
  il catalogo dei design pattern non è stato derivato da considerazioni teoriche ma proviene dall'esperienza "sul campo" di progettisti sw: si tratta di soluzioni collaudate a problemi tipici della cui applicazione si conoscono benefici e limitazioni
- **Riconoscere al volo un problema di progettazione**
  i design pattern mantengono un impianto generico permettendone l'applicabilità a *classi* di problemi: se li conosciamo in anticipo faremo molta meno fatica (e perderemo meno tempo) a risolvere problemi che hanno una struttura simile
- **Rendere più chiaro un progetto**
  i design pattern rappresentano spesso un "vocabolario" comune tra progettisti software. nominarli/individuarli consente di risparmiare molti sforzi di comunicazione -> è bene riferirli nel proprio progetto qualora se ne faccia uso
- **Qualità del progetto**
  Un corretto utilizzo dei design pattern rende il progetto (e il codice !) più snello e comprensibile, favorisce il riuso del codice e la sua manutenzione (convenzioni, refactoring, estendibilità)

Design Patterns

- Testo fondamentale:
  E. Gamma, R. Helm, R. Johnson, J. Vlissides (gang of 4), *Design Patterns – Elements of Reusable Object-Oriented Software*

- In questo testo è riportato il *catalogo* dei pattern (23 pattern diversi) ed un caso di studio sulla loro applicazione (editor di documenti Lexi)

Catalogo dei Pattern

Per ogni pattern sono riportati:

- Pattern Name and Classification
- Intent
- Also Known As
- Motivation
- Applicability
- ***Structure*** ⇨ *rappresentazione grafica delle classi del pattern usando una notazione basata sulla Object Modeling Technique (OMT)*
- Participants (le classi coinvolte)
- Collaborations (ruolo, tipologia di comunicazione)
- Consequences
- Implementation (molti design pattern prevedono diverse scelte implementative)
- Sample Code
- Known Uses
- Related Patterns

Categorie: Purpose

3 categorie di pattern
in base alla funzione (**purpose**):

- **Creazionali (Creational)**
  forniscono meccanismi per la creazione di oggetti
- **Strutturali (Structural)**
  gestiscono la separazione tra interfaccia e implementazione e le modalità di composizione tra oggetti per creare strutture dati complesse
- **Comportamentali (Behavioral)**
  consentono la modifica del comportamento degli oggetti minimizzando la quantità di codice da cambiare

Categorie: Scope

2 categorie di pattern
in base al dominio (**scope**):

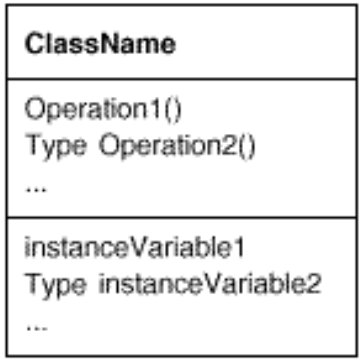
- **Class pattern**
  - si focalizzano sulle relazioni tra classi e sottoclassi
  - tipicamente si riferiscono a situazioni statiche (per es., relazioni espresse attraverso l'ereditarietà)

- **Object pattern**
  - si focalizzano su oggetti (istanze di classi) e loro relazioni
  - tipicamente si riferiscono a situazioni dinamiche (le relazioni tra oggetti possono ovviamente cambiare a run-time). Uso della "composizione" tra oggetti.

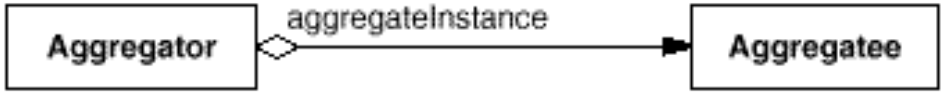
Categorie

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

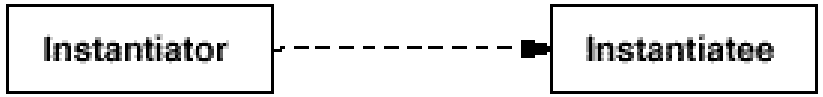
Interpretazione dei Pattern



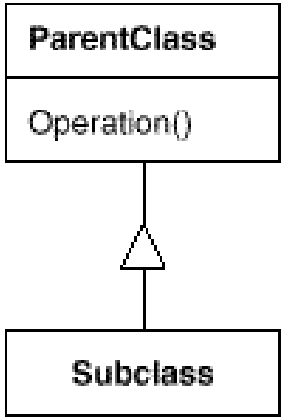
Classe



Aggregazione

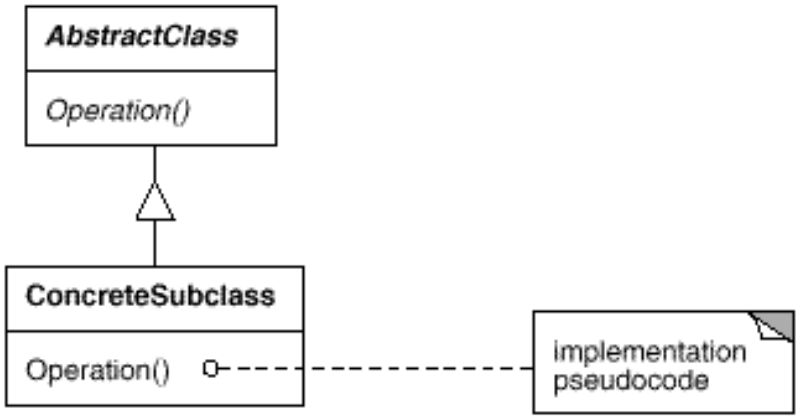


Indica una classe che istanzia oggetti di un'altra classe

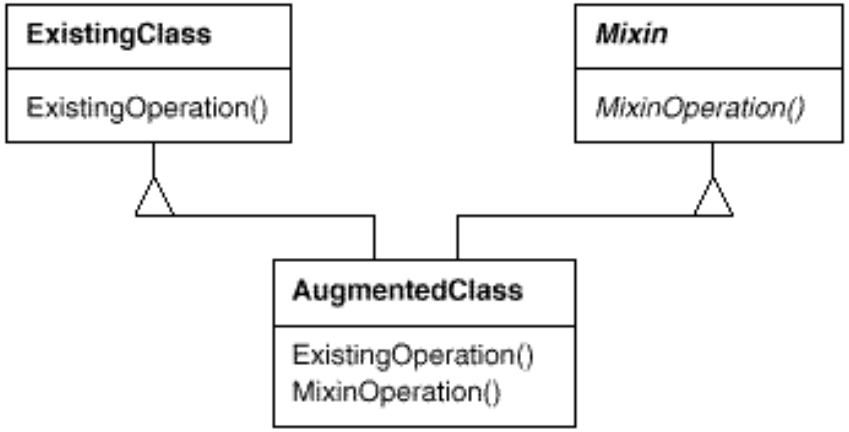


Ereditarietà

Interpretazione dei Pattern (ii)



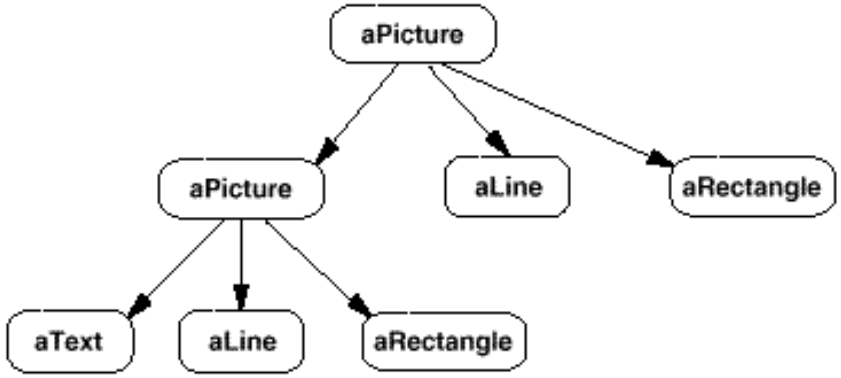
Classe astratta



Ereditarietà multipla

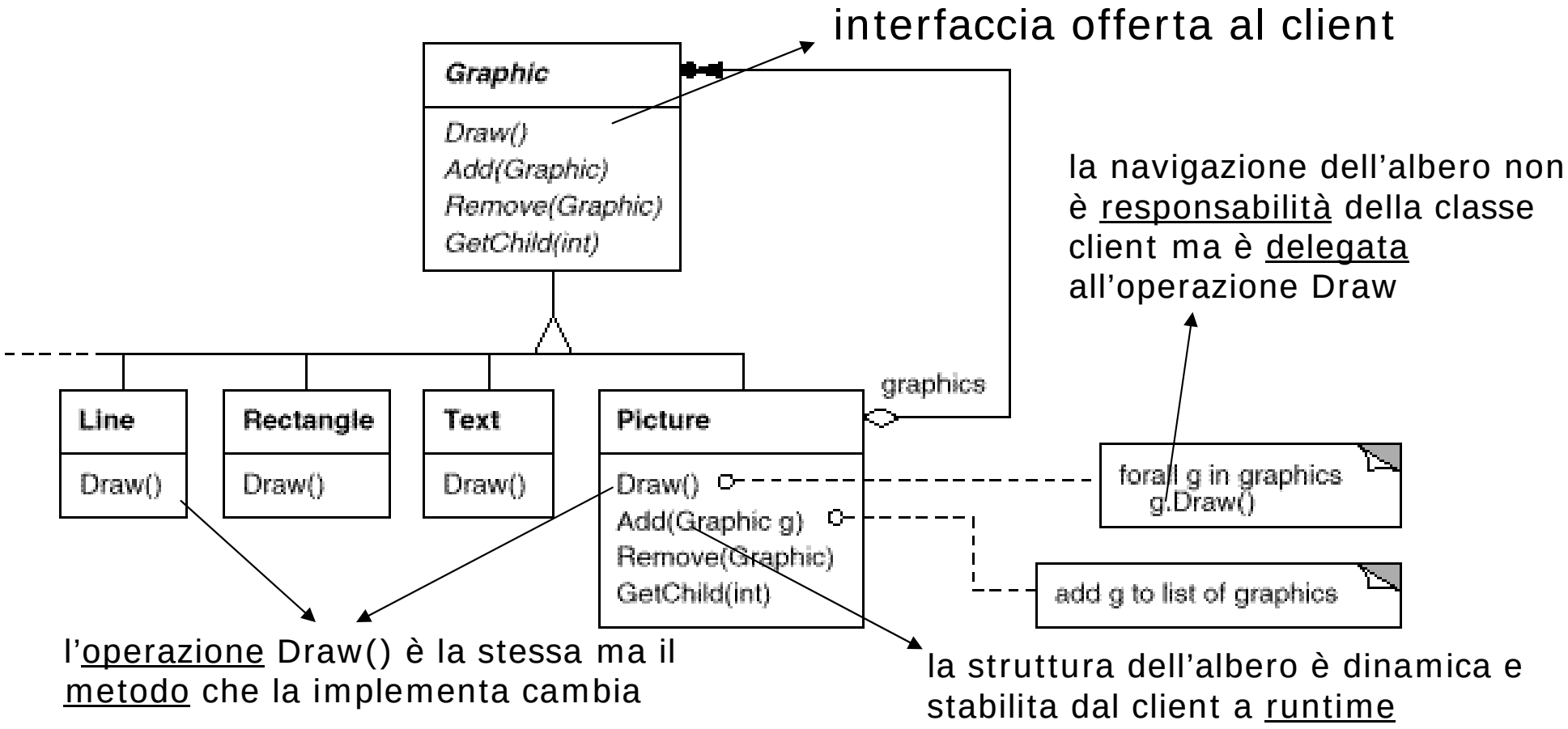
Esempio : Composite

- Un elemento grafico (Graphic) può essere una linea (Line), un rettangolo (Rectangle), una casella di testo (Text) o una figura (Picture)
- Una figura (Picture) può contenere a sua volta uno o più elementi grafici:



- Ciascun elemento grafico deve supportare l'operazione draw(): si vuole rendere il più possibile conforme l'interfaccia di un oggetto composto a quella di un oggetto semplice. Ciò vuol dire che una classe client associata ad un oggetto di tipo Graphic deve poterlo utilizzare senza preoccuparsi se si tratta di una Picture o di una casella di testo.

Composite – Soluzione

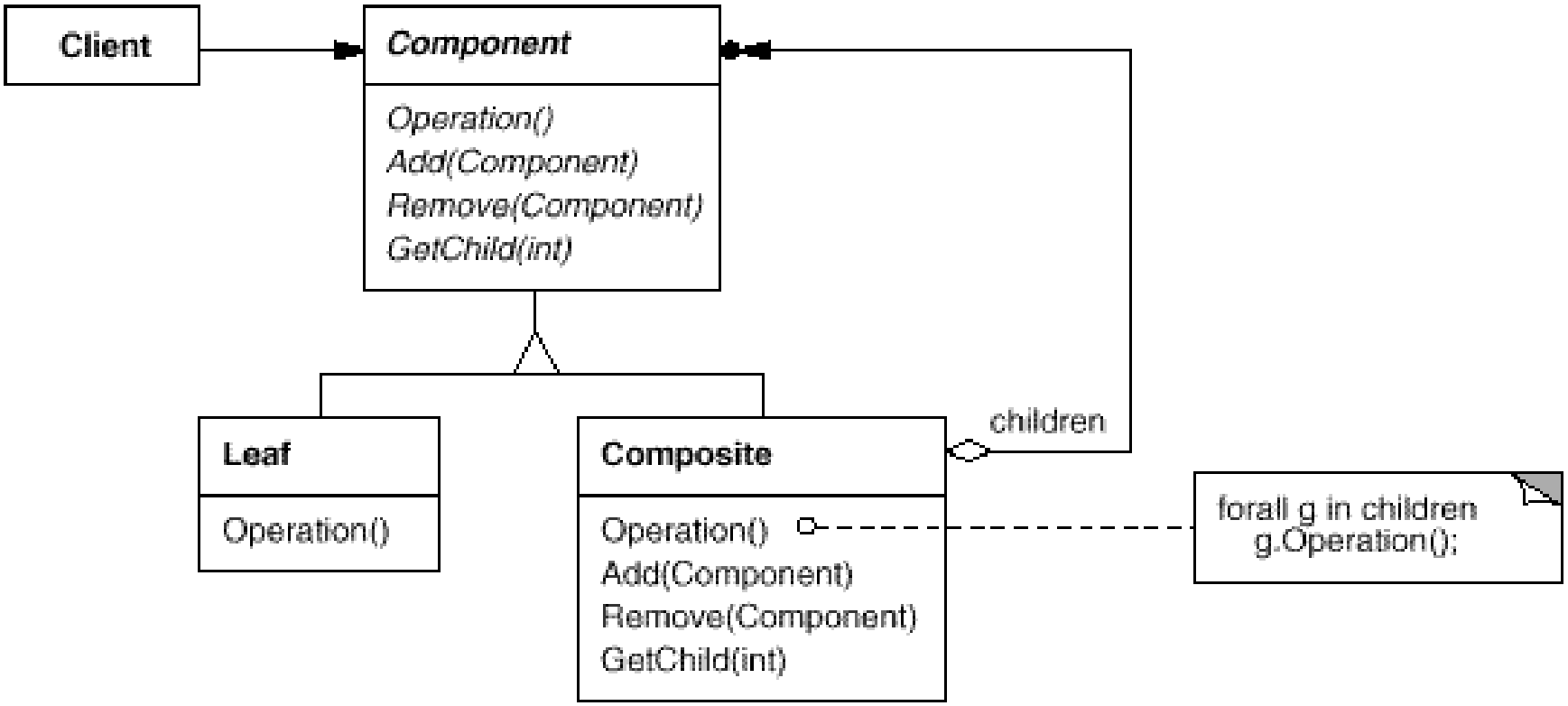


quali sono i meccanismi orientati agli oggetti offerti da un linguaggio come il C++ per implementare il pattern Composite?

Composite (object/structural)

- Obiettivi:
  - Fornire la possibilità di comporre oggetti in strutture ad albero che rappresentano gerarchie *intero-parte*
  - Consentire ai client di trattare oggetti singoli e compositi in modo uniforme
  - Minimizzare il più possibile la complessità di una gerarchia *intero-parte*
    - Ridurre il numero di tipologie di oggetti che possono trovarsi nei diversi nodi dell'albero

Composite (ii)



(generalizzazione dell'esempio precedente)

Composite (iii)

- Consente di definire gerarchie di classi costituite da oggetti primitivi e compositi
  - Gli oggetti primitivi possono essere composti per formare oggetti più complessi, che a loro volta potranno essere composti ricorsivamente
  - In tutti i punti in cui il client si aspetta di utilizzare un oggetto primitivo, potrà essere indifferentemente utilizzato un oggetto composito

- Semplifica il client
  - I client possono trattare strutture composite e singoli oggetti in modo uniforme
  - I client solitamente non sanno (e non dovrebbero neanche preoccuparsene) se stanno operando con una foglia o un componente composito

Riferimenti

[Gamma95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1995

[Pianciamore03] M. Pianciamore, *Design Pattern*, slides del corso di “Ingegneria del Software II”, A.A. 2003/2004