



UNIVERSITÀ DEGLI STUDI DI SALERNO
Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea in Informatica

Object Design: riuso e flessibilità

Ereditarietà, Composizione e Design Patterns

Corso di Ingegneria del Software

Object Design (3)

- L' **Object Design** include:

- **Riuso:** impiego dell' ereditarietà, di componenti off-the-shelf e dei design pattern per poter riutilizzare soluzioni esistenti.
- **Specifica dei servizi:** descrizione precisa delle interfacce di ogni classe
- **Ristrutturazione del modello ad oggetti:** il modello di design object viene trasformato per migliorare la sua comprensibilità ed estensibilità
- **Ottimizzazione del modello ad oggetti:** trasformiamo il modello di design object in modo tale da soddisfare criteri di prestazione, quali tempo di risposta o utilizzo della memoria.

Oggetto di
questa lezione

Attività Object Design: Riuso

- **Obiettivi dell' attività di riuso:**
 - Utilizzare funzionalità già disponibili per realizzare nuove funzionalità.
 - Utilizzare la conoscenza acquisita durante precedenti esperienze di progettazione per risolvere il problema corrente.
- **Strumenti impiegati:**
 - **Ereditarietà:** le nuove funzionalità sono ottenute per ereditarietà.
 - detta anche riuso White-Box perchè la struttura interna delle classi antenate è spesso visibile alle sottoclassi;
 - **Composizione:** le nuove funzionalità sono ottenute per aggregazione
 - detta anche riuso Black-Box perché i dettagli implementativi interni agli oggetti non sono visibili all' esterno;
 - **Design Pattern:** template di soluzioni a problemi ricorrenti impiegati per ottenere riuso e flessibilità.

Attività Object Design: Riuso

- **Obiettivi dell' attività di riuso:**

- Utilizzare funzionalità già disponibili per realizzare nuove funzionalità.
- Utilizzare la conoscenza acquisita durante precedenti esperienze di progettazione per risolvere il problema corrente.

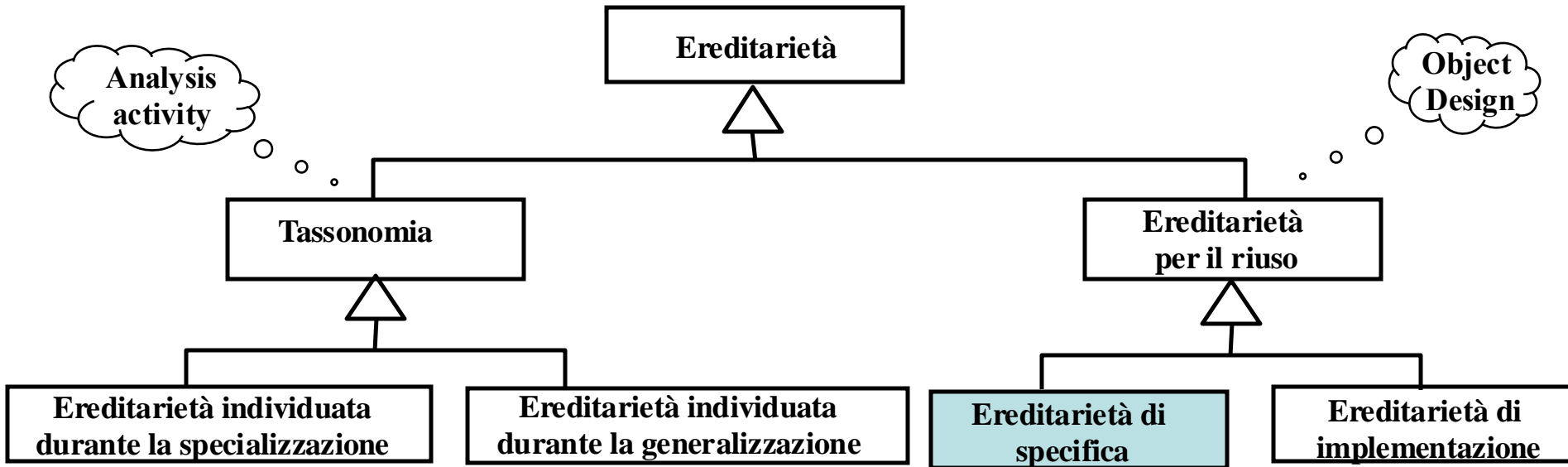
- **Strumenti impiegati:**

- **Ereditarietà:** le nuove funzionalità sono ottenute per ereditarietà.
 - detta anche riuso White-Box perchè la struttura interna delle classi antenate è spesso visibile alle sottoclassi;
- **Composizione:** le nuove funzionalità sono ottenute per aggregazione
 - detta anche riuso Black-Box perché i dettagli implementativi interni agli oggetti non sono visibili all' esterno;
- **Design Pattern:** template di soluzioni a problemi ricorrenti impiegati per ottenere riuso e flessibilità.

Ereditarietà: Requirements Analysis e Object Design

- Durante l' **analisi** abbiamo usato l' **ereditarietà** per organizzare gli oggetti in una **gerarchia comprensibile (descrizione di tassonomie)**.
 - Partendo dagli oggetti astratti i lettori del modello di analisi comprendono le funzionalità chiave del sistema.
- L' utilizzo dell' **ereditarietà** durante l' **object design** permette di **ridurre le ridondanze, migliorare l' estendibilità, la modificabilità e il riuso del codice**.
 - I comportamenti ridondanti sono fattorizzati in una singola superclasse, riducendo il rischio di introdurre inconsistenze in seguito a futuri cambiamenti.
 - Ereditarietà di specifica
 - Ereditarietà di implementazione
- Bisogna però stare **attenti** poiché l' **ereditarietà** è un **meccanismo molto potente**.
 - Sviluppatori inesperti spesso producono codice meno comprensibile e più fragile di quello che otterrebbero senza farne uso.

Ereditarietà di specifica



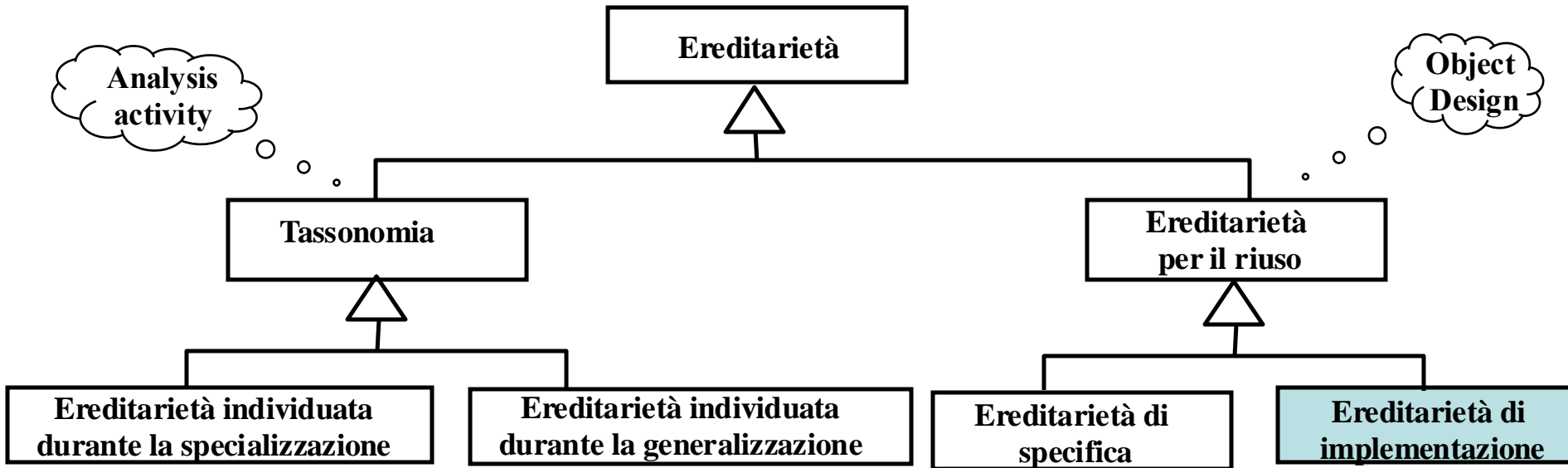
- L' **Ereditarietà di specifica** (o di interfaccia, o subtyping) definisce la possibilità di utilizzare un oggetto al posto di un altro
 - si eredita da una classe astratta che possiede operazioni già specificate ma non implementate

Ereditarietà di specifica: il principio di sostituzione di Liskov

“Se un oggetto di tipo S può essere sostituito ovunque ci si aspetta un oggetto di tipo T, allora S è un sottotipo di T”

- Fornisce una definizione formale per l' eredità di specifica.
- Afferma che un metodo scritto in termini di una superclasse T deve essere in grado di usare istanze di qualunque sottoclasse di T senza sapere se utilizza la superclasse o una sua sottoclasse.
 - gli oggetti appartenenti a una sottoclasse devono essere in grado di esibire tutti i comportamenti e le proprietà esibiti da quelli appartenenti alla superclasse in modo tale da poter essere "sostituiti" senza intaccare la funzionalità del programma
 - si noti che non si richiede che la sottoclasse abbia solo le caratteristiche esibite dalla superclasse
- Una relazione di ereditarietà che soddisfa tale principio è chiamata Ereditarietà stretta.
- Esempio: data una classe telefono deriviamo la sottoclasse cellulare
 - affinché la classe cellulare possa essere concepita come sottoclasse di telefono occorre che un cellulare possa essere usato in tutti i contesti in cui si richiede l'uso di un telefono
 - il fatto che un cellulare possa anche inviare SMS non inficia il fatto che esso sia sostituibile a un telefono

Ereditarietà di implementazione



- L' **Ereditarietà di implementazione (o di classe)** definisce l' implementazione di un oggetto in funzione dell' implementazione di un altro oggetto
 - si eredita da una classe esistente che possiede tutte le operazioni già implementate

Ereditarietà di implementazione: esempio

- Dobbiamo realizzare la struttura dati stack.

Stack
push(Object elem) pop() Object top()

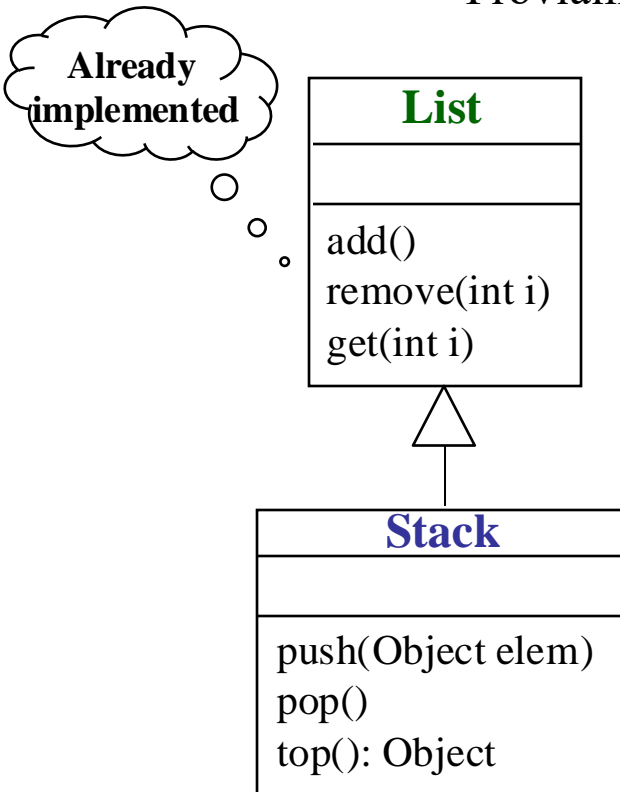
- Abbiamo a disposizione l'implementazione della classe List che svolge operazioni simili a quelle necessarie per realizzare uno stack.

List
add() remove(int i) get(int i)

Ereditarietà di implementazione: esempio

- Come possiamo **riutilizzare** l'implementazione della classe List per realizzare la classe Stack?

Proviamo ad utilizzare l'ereditarietà di implementazione.



```
/* Implementation of Stack using inheritance*/  
  
Class Stack extends List {  
    //Constructor omitted  
    Object top() {  
        return get(this.size()-1);  
    }  
    void push (Object elem){  
        add(elem);  
    }  
    void pop() {  
        remove(this.size()-1);  
    }  
}
```

Ereditarietà di implementazione: considerazioni

- Un problema con l'ereditarietà di implementazione:
 - Le operazioni della superclasse sono esposte all'utilizzatore della sottoclasse.
 - Alcune operazioni ereditate possono essere utilizzate in modo inaspettato:
 - cosa succederebbe se l'utilizzatore della classe Stack invocasse remove() invece di pop()?

```
/* using a Stack object*/  
public static void main(String args[]) throws IOException {  
    Stack aStack=new Stack();  
    aStack.remove(3);  
}
```

- Non viene segnalato nessun errore, né in fase di compilazione né in fase di esecuzione.
- È evidente però che è presente un errore logico: si utilizza una struttura LIFO in modo scorretto.

Attività Object Design: Riuso

- **Obiettivi dell' attività di riuso:**

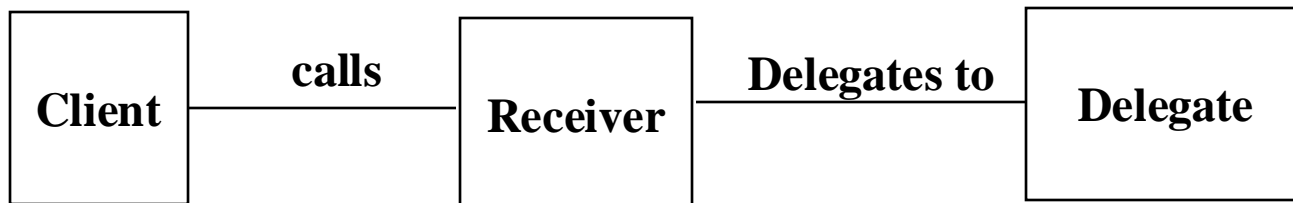
- Utilizzare funzionalità già disponibili per realizzare nuove funzionalità.
- Utilizzare la conoscenza acquisita durante precedenti esperienze di progettazione per risolvere il problema corrente.

- **Strumenti impiegati:**

- **Ereditarietà:** le nuove funzionalità sono ottenute per ereditarietà.
 - detta anche riuso White-Box perchè la struttura interna delle classi antenate è spesso visibile alle sottoclassi;
- **Composizione:** le nuove funzionalità sono ottenute per aggregazione
 - detta anche riuso Black-Box perché i dettagli implementativi interni agli oggetti non sono visibili all' esterno;
- **Design Pattern:** template di soluzioni a problemi ricorrenti impiegati per ottenere riuso e flessibilità.

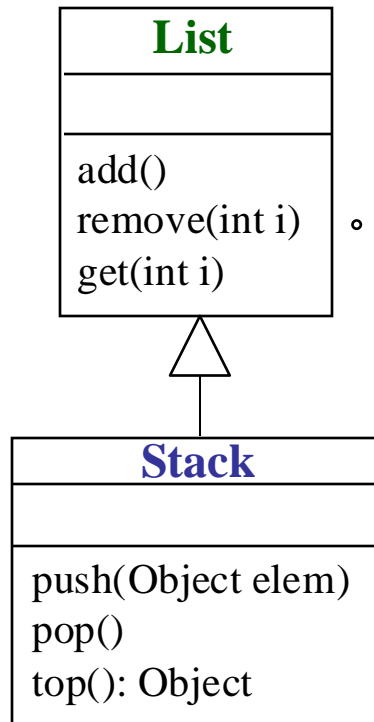
Un esempio di composizione: la delega

- La **delega** è un' alternativa altrettanto efficace all' ereditarietà di implementazione.
- Due oggetti collaborano nel gestire una richiesta:
 - Un **Client** richiede l' esecuzione di un' operazione all' oggetto **Receiver**
 - **Receiver** delega ad un altro oggetto (**Delegate**) l' esecuzione dell' operazione
 - In questo modo l' oggetto delegato non può essere utilizzato in modo scorretto dall' utilizzatore dell' oggetto ricevente.

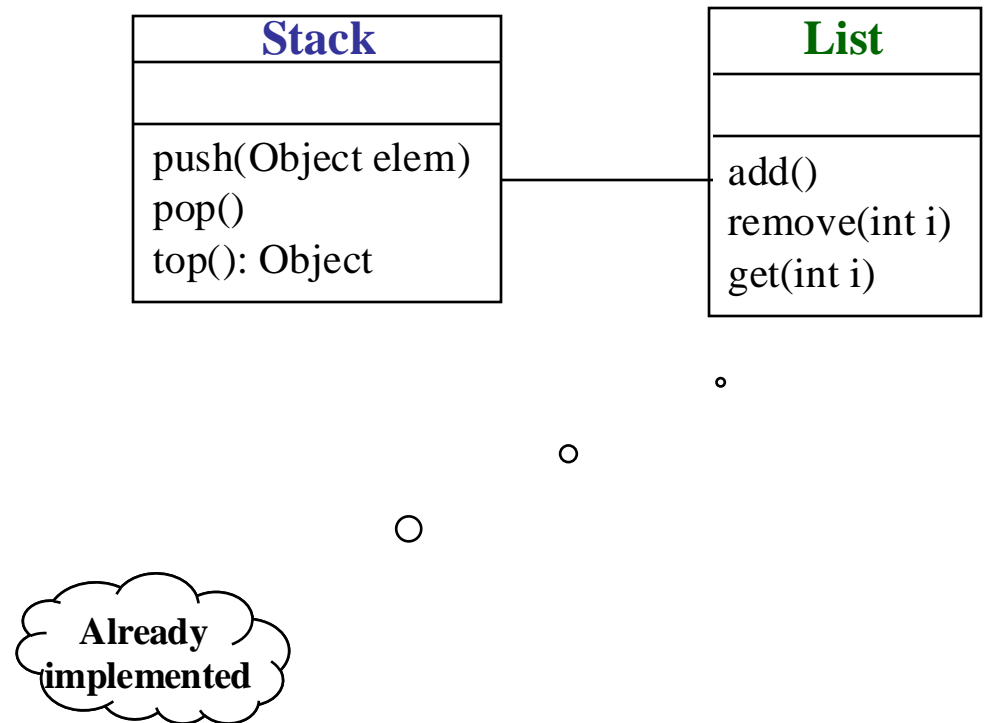


Delega: esempio

Ereditarietà di implementazione



Delega



Delega: esempio

Ereditarietà di implementazione

```
/* Implementation of Stack using inheritance*/
```

```
class Stack extends List {  
  
    //Constructor omitted  
  
    Object top() {  
        return get(this.size()-1);  
    }  
  
    void push (Object elem){  
        add(elem);  
    }  
  
    void pop() {  
        remove(this.size()-1);  
    }  
}
```

Delega

```
/* Implementation of Stack using delegation*/
```

```
class Stack {  
    private List aList;  
  
    Stack() {  
        aList = new List();  
    }  
  
    Object top() {  
        return aList.get(aList.size()-1);  
    }  
  
    void push (Object elem) {  
        aList.add(elem);  
    }  
  
    void pop () {  
        aList.remove(aList.size()-1);  
    }  
}
```

Delega: considerazioni

- Usando la delega la classe **Stack** non include nella sua interfaccia i metodi di **List** ed il nuovo campo `aList` è privato:
 - l'utente (client) della classe `Stack` non può utilizzare i metodi di `List` (non si può sfruttare il subtyping);
 - possiamo cambiare la rappresentazione interna di `Stack` con un'altra classe (ad esempio `Vector`) senza influenzare i client di `Stack`;
- La delega non interferisce con le componenti esistenti e porta allo sviluppo di codice più robusto.
- L'ereditarietà di specifica è preferibile alla delega in situazioni di subtyping poiché porta a design maggiormente estendibili.

Delega o Ereditarietà di implementazione?

- **Delega**

- **Pro:**

- L'incapsulamento non è violato: si accede agli oggetti solo attraverso la loro interfaccia.
 - Flessibilità: Consente di comporre facilmente comportamenti in fase di esecuzione e di cambiare il modo in cui questi comportamenti sono composti.

- **Contro:**

- È definita dinamicamente durante l'esecuzione attraverso oggetti che acquisiscono riferimenti ad altri oggetti: più inefficiente in esecuzione rispetto a software statico.

- **Ereditarietà**

- **Pro:**

- Definita staticamente al momento della compilazione.
 - Immediata da utilizzare (è supportata direttamente dal linguaggio di programmazione).

- **Contro:**

- È impossibile cambiare l'implementazione ereditata durante l'esecuzione.
 - L'implementazione di una sottoclasse diventa strettamente dipendente dalla classe padre infatti qualsiasi modifica nell'implementazione della classe padre induce modifiche nella sottoclasse (devono essere ricomilate entrambe).

Attività Object Design: Riuso

- **Obiettivi dell' attività di riuso:**

- Utilizzare funzionalità già disponibili per realizzare nuove funzionalità.
- Utilizzare la conoscenza acquisita durante precedenti esperienze di progettazione per risolvere il problema corrente.

- **Strumenti impiegati:**

- **Ereditarietà:** le nuove funzionalità sono ottenute per ereditarietà.
 - detta anche riuso White-Box perchè la struttura interna delle classi antenate è spesso visibile alle sottoclassi;
- **Composizione:** le nuove funzionalità sono ottenute per aggregazione
 - detta anche riuso Black-Box perché i dettagli implementativi interni agli oggetti non sono visibili all' esterno;

- **Design Pattern:** template di soluzioni a problemi ricorrenti impiegati per ottenere riuso e flessibilità.

Ottenere riuso e flessibilità: Design patterns

“Catturare l’esperienza e la saggezza degli esperti al fine di evitare di reinventare ogni volta le stesse cose”

- Oltre al riuso nella scrittura del software esiste anche un riuso nella progettazione.
- L’esperienza è un fattore fondamentale per una buona progettazione
 - un progettista esperto non parte mai da zero, riutilizza soluzioni che si sono dimostrate valide in passato.
- La “comunità dei pattern” si propone come obiettivo la catalogazione di questi schemi ricorrenti in modo da costituire un dizionario a disposizione dei progettisti.

Ottenere riuso e flessibilità: Design patterns

“Strict modeling of the real world leads to a system that reflects today’ s realities but not necessarily tomorrow’ s”

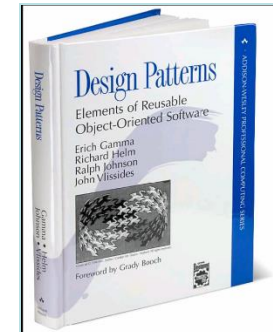
- Vorremmo progettare prodotti **software** facilmente **modificabili ed estendibili** per minimizzare il costo di modifiche future.
- Le **fonti che inducono alla modifica** di un software possono essere **molteplici** e spesso **comuni alla maggior parte dei prodotti software**:
 - **Nuovo produttore o nuova tecnologia** (Le componenti commerciali utilizzate per costruire il sistema sono spesso sostituite da altre equivalenti da un differente produttore).
 - **Nuova implementazione** (Quando i sottosistemi sono integrati e testati insieme, il tempo di risposta dell’ intero sistema è superiore alle prestazioni richieste).
 - **Nuove viste** (Testare il software con utenti reali rivela molti problemi di usabilità. Questo porta alla creazione di diverse viste sugli stessi dati).
 - **Nuova complessità del dominio di applicazione** (Il rilascio di un sistema causa idee di nuove generalizzazioni. Es: numero di volo, inizialmente unico. Unificazione di compagnie: si può avere lo stesso numero per più compagnie).
 - **Errori** (Molti errori nei requisiti sono scoperti quando gli utenti reali iniziano ad usare il sistema)
- Una **soluzione** è **prevedere i cambiamenti e tenerne conto nella progettazione del sistema** in quanto i cambiamenti tendono ad essere simili per molti sistemi.

Cos' è un pattern?

“Ogni pattern descrive un **problema** che **ricorre** più volte nel nostro **ambiente**, descrive poi il **nucleo** della **soluzione** del problema in modo da poter **utilizzare** tale soluzione un **milione** di volte senza **mai** farlo allo stesso modo”

Christopher Alexander

- Idea presa in prestito dal campo dell'architettura
 - C. Alexander *et al.*, “A pattern language: towns, buildings, construction”, 1977.
- Ripresa e adattata alla progettazione del software:
 - Gamma *et al.*, “Design Patterns: Elements of Reusable Object-Oriented Software”, 1994.
 - Gli autori sono noti anche come Gang of Four (GoF).
 - Il libro descrive 23 pattern che vengono chiamati GoF Patterns, in seguito ne sono stati individuati e descritti molti altri.



Design patterns

- Nello sviluppo OO, i **design pattern** sono **template di soluzioni** (che gli sviluppatori hanno raffinato nel tempo) utilizzabili per **risolvere** un insieme di **problemi ricorrenti**
 - Es. separare l'interfaccia da un numero variabile di implementazioni, incapsulare un insieme di classi legacy, proteggere un client da modifiche associate a piattaforme specifiche.
- Un design pattern è **solitamente composto da poche classi**
 - **correlate** tramite **delegazione** ed **ereditarietà** per fornire una soluzione robusta e modificabile.
- Tali **classi** possono essere **adattate e ridefinite** per lo **specifico sistema** che si vuole realizzare (*personalizzazione del sistema; riuso di soluzioni esistenti*).
- **N.B.:** i design pattern non spiegano come modellare liste a puntatori o hashmap, né trattano progetti complessi relativi ad intere applicazioni o sottosistemi, ma descrivono oggetti comunicanti e classi, adattate in maniera tale da risolvere specifici problemi di progettazione.

Com'è fatto un design pattern?

- Un *design pattern* è definito mediante 4 elementi principali:
 - Un *nome* simbolico che lo identifica.
 - Una *descrizione del problema* che descrive le situazioni in cui il pattern può essere utilizzato.
 - Una *soluzione* che descrive gli elementi che costituiscono il progetto, le loro relazioni, responsabilità e collaborazioni.
 - Un insieme di *conseguenze* che descrivono i risultati e i vincoli che si ottengono applicando il pattern. Sono utili per valutare i trade-off e le alternative che devono essere considerate rispetto ai design goal affrontati.
- A questo *schema di base* possono corrispondere *schemi più dettagliati* (per esempio quello GoF).

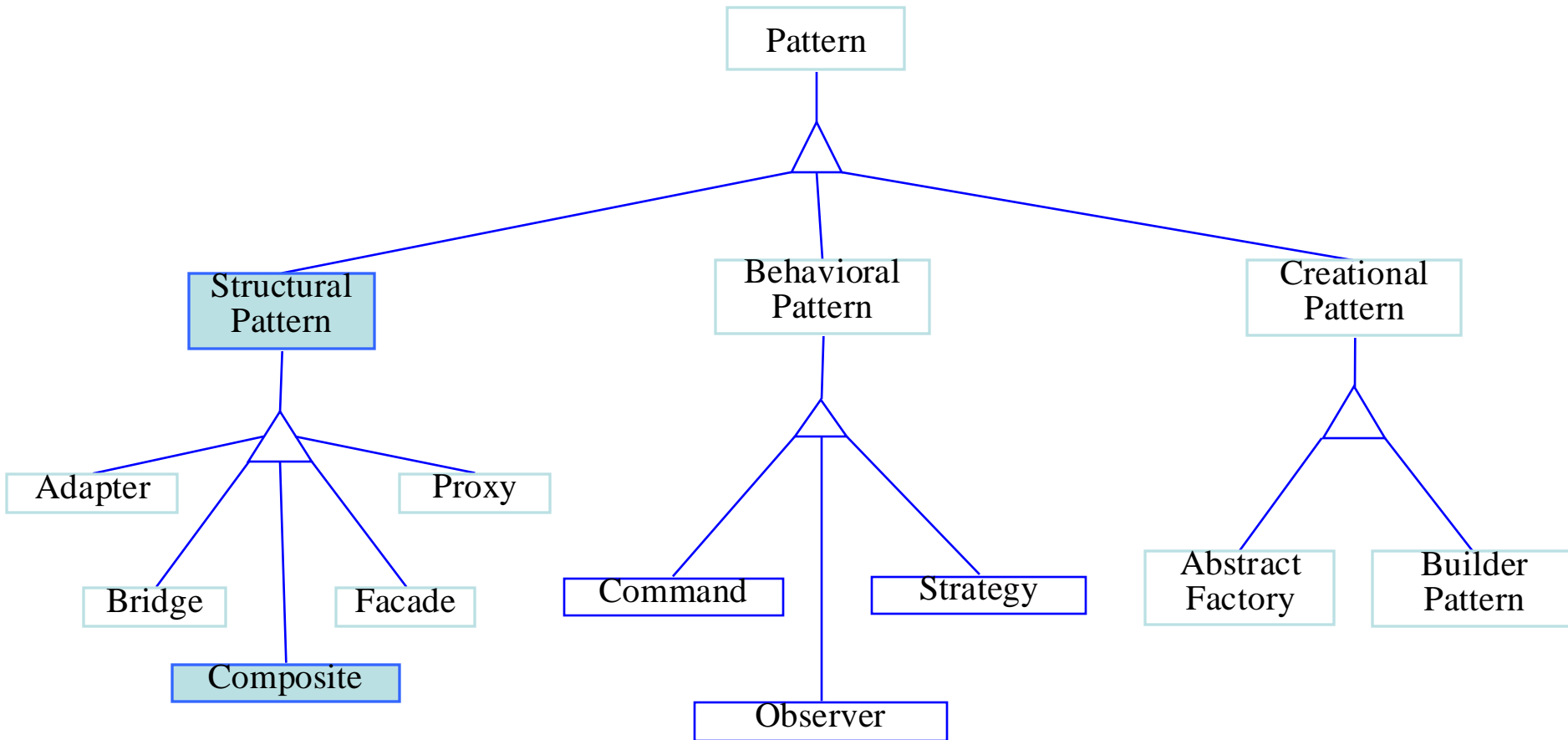
Struttura di un pattern: schema di descrizione GoF

- **Nome e classificazione**: importante per il vocabolario.
- **Scopo**: cosa fa il pattern e suo fondamento logico.
- **Nomi alternativi**: molti pattern sono conosciuti con più nomi.
- **Motivazione**: scenario che illustra un problema di progettazione e la soluzione offerta.
- **Applicabilità**: quando può essere applicato il pattern.
- **Struttura (o modello)**: rappresentazione UML del pattern.
- **Partecipanti**: le classi/oggetti con le proprie responsabilità.
- **Collaborazioni**: come collaborano i partecipanti.
- **Conseguenze**: pro e contro dell'applicazione del pattern.
- **Implementazione**: come si può implementare il pattern.
- **Codice d'esempio**: frammenti di codice.
- **Pattern correlati**: relazioni con altri pattern.
- **Utilizzi noti**: esempi di utilizzo reale del pattern in sistemi esistenti.

Classificazione dei design pattern (GoF)

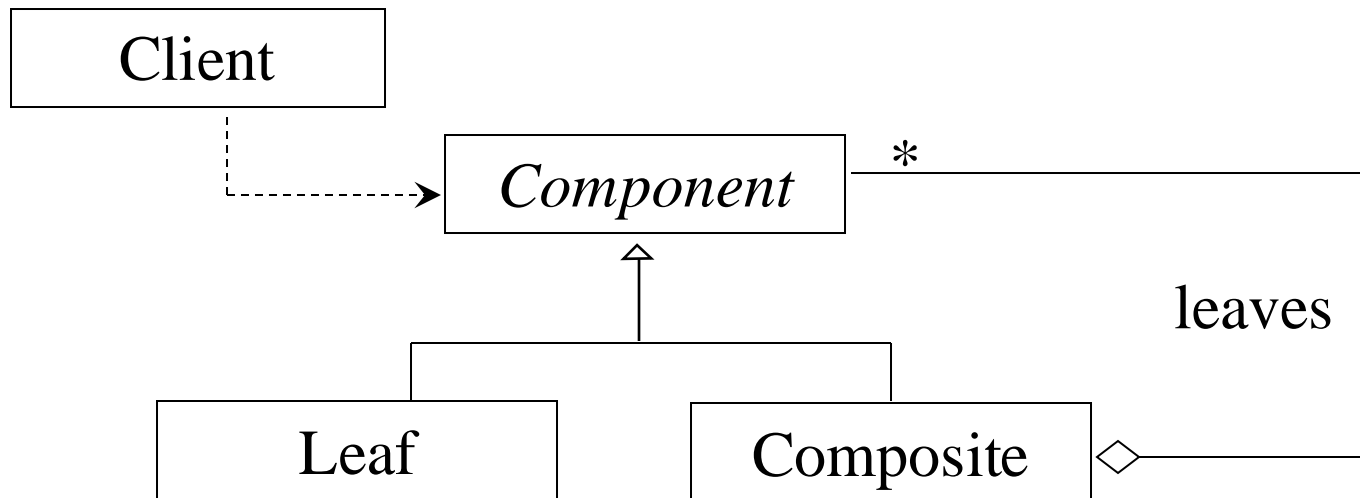
- **Pattern strutturali (structural patterns)**
 - Si occupano delle modalità di composizione di classi e oggetti per formare strutture complesse.
 - Riducono l'accoppiamento tra due o più classi, incapsulano strutture complesse, introducono una classe astratta per permettere estensioni future.
- **Pattern comportamentali (behavioral patterns)**
 - Si occupano di algoritmi e dell'assegnazione delle responsabilità tra oggetti che collaborano tra loro (*chi fa che cosa?*).
 - Caratterizzano i flussi di controllo complessi difficili da seguire a run time.
- **Pattern creazionali (creational patterns)**
 - Forniscono un'astrazione del processo di creazione degli oggetti.
 - Aiutano a rendere un sistema indipendente dalle modalità di creazione, composizione e rappresentazione degli oggetti utilizzati.

Pattern strutturali: Composite Pattern



Composite Pattern (1)

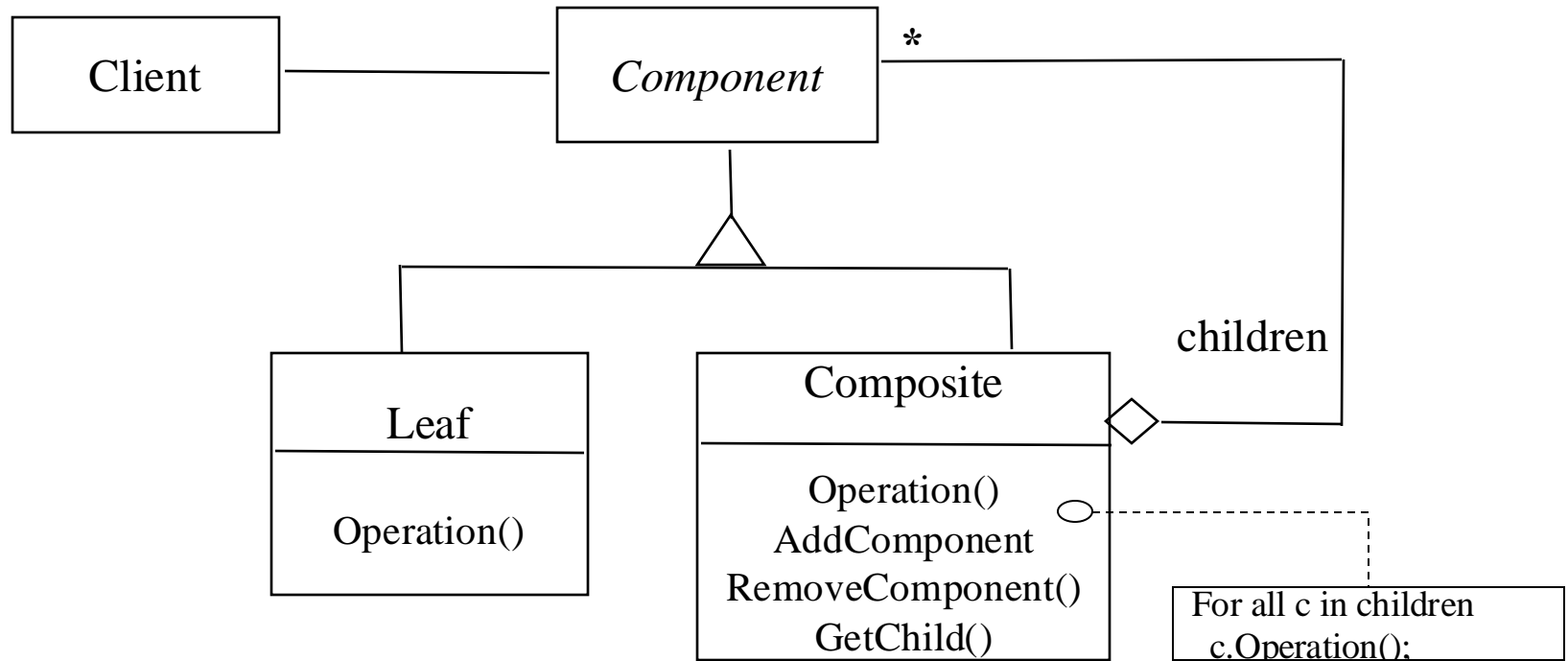
- **Nome:** Composite
- **Descrizione del problema:**
 - compone gli oggetti in strutture ad albero di ampiezza e profondità variabile per rappresentare gerarchie tutto-parte;
 - permette al client di trattare gli oggetti individuali (foglie) e gli oggetti composti (nodi interni) in maniera uniforme attraverso un'interfaccia comune.



Composite Pattern (2)

- **Soluzione:**

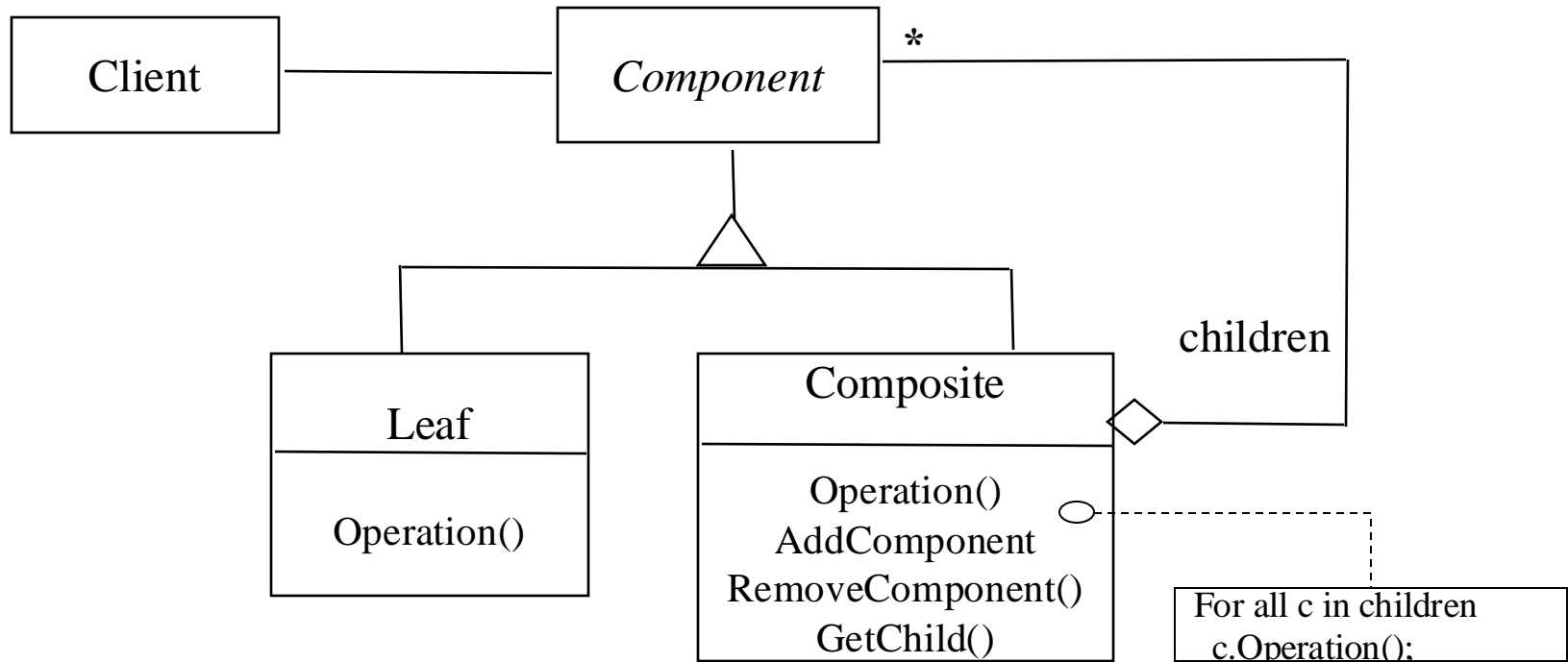
- l'interfaccia *Component* specifica i servizi che sono condivisi tra *Leaf* e *Composite*;
- un *Composite* ha un'associazione di aggregazione con *Component* e implementa ogni servizio iterando su ogni *Component* che contiene.
- i servizi *Leaf* fanno il lavoro effettivo.



Composite Pattern (3)

- **Conseguenze:**

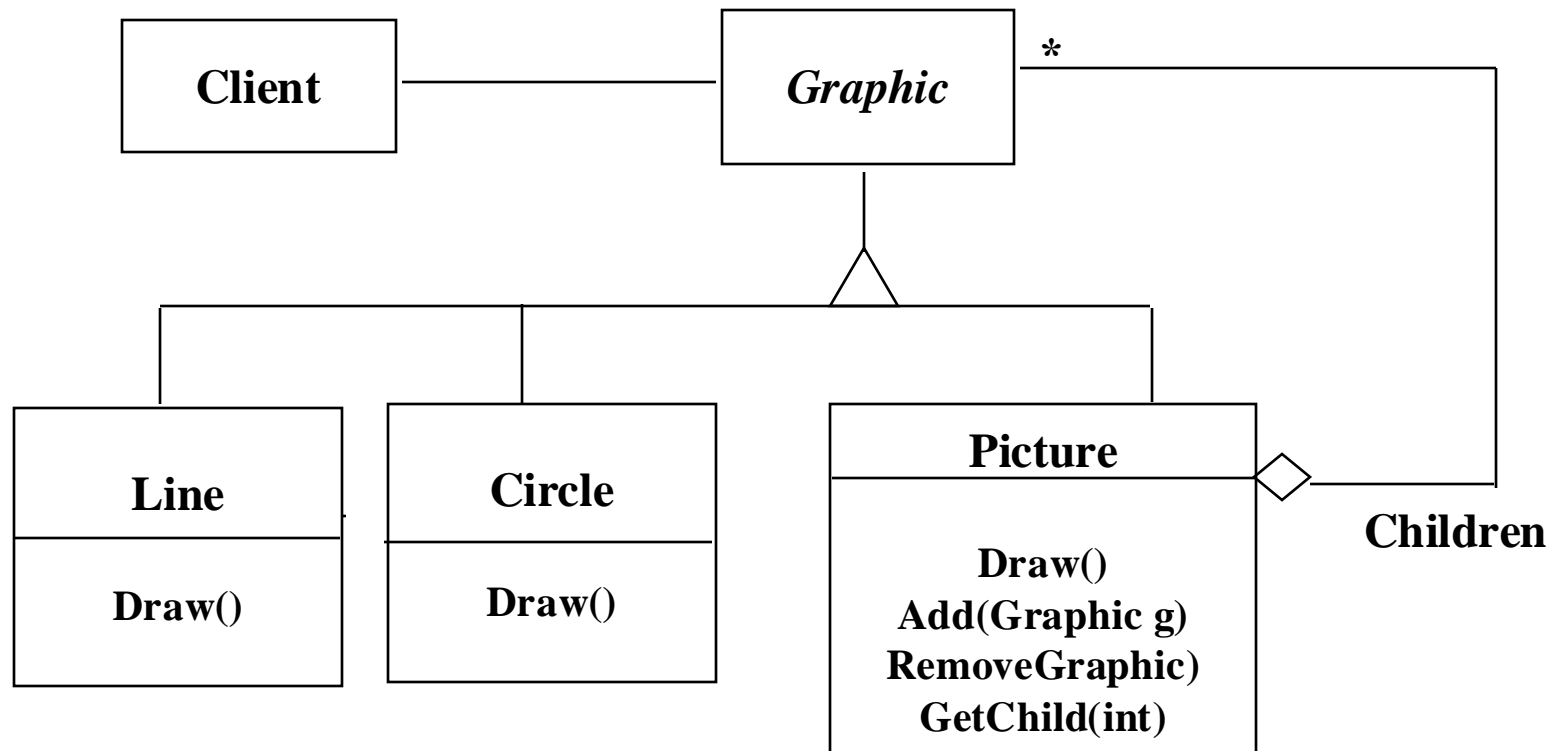
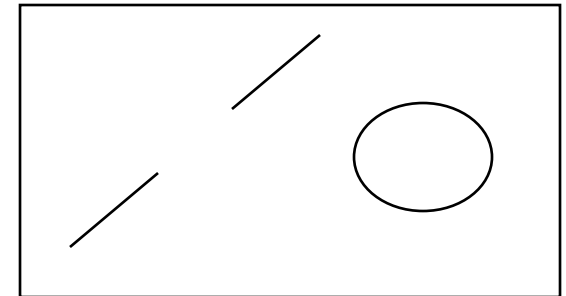
- il Client usa lo stesso codice per gestire i Leaf o i Composite;
- i comportamenti specifici di un Leaf possono essere cambiati senza cambiare la gerarchia;
- nuove classi di Leaf possono essere aggiunte senza cambiare la gerarchia.



Composite Pattern (4)

- **Esempio:**

- le applicazioni grafiche usano il pattern composite



Composite Pattern: esempio (1)

- Il pattern composite permette di modellare strutture mediante la composizione di classi e oggetti .
- Toolkit per interfacce grafiche, come Swing, forniscono agli sviluppatori un insieme di classi come blocchi predefiniti:
 - l' interfaccia utente può essere costruita aggregando queste componenti.
- I moderni toolkit gestiscono la complessità di dover gestire molti oggetti organizzando gli oggetti in **gerarchie** di nodi aggregati chiamati panel:
 - i panel possono essere manipolati allo stesso modo degli oggetti concreti dell' interfaccia.
- Le Swing risolvono questo problema utilizzando il pattern Composite.

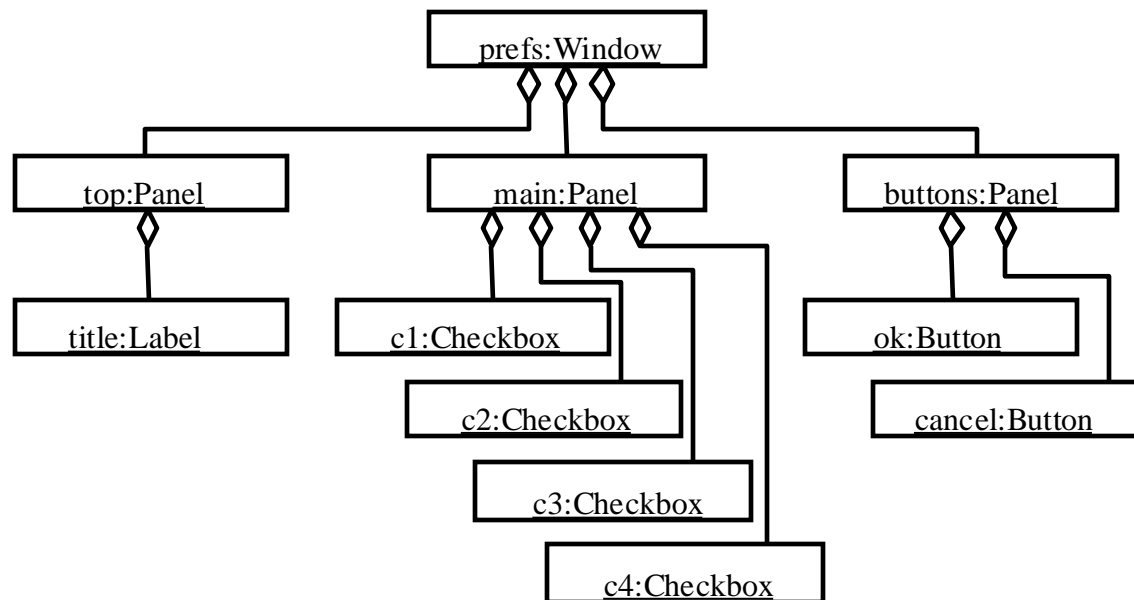
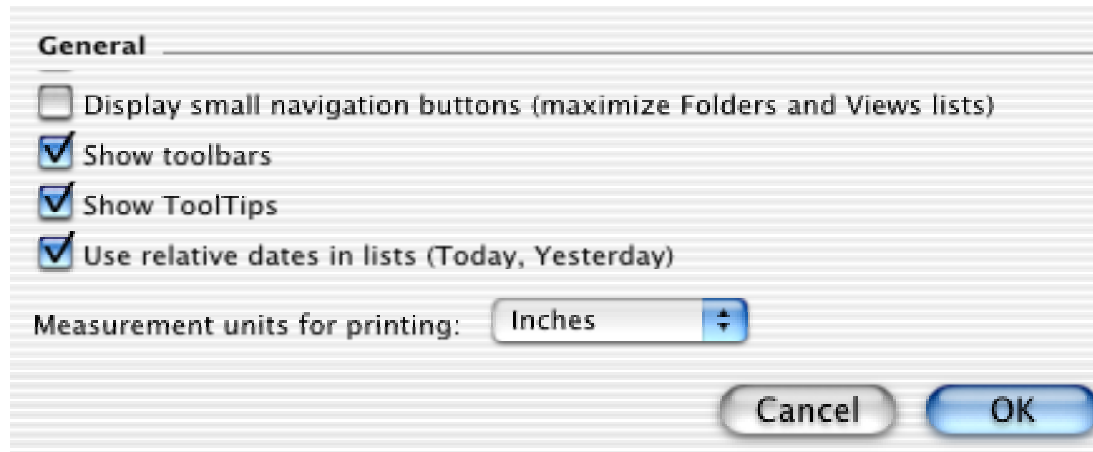
Composite Pattern: esempio (2)

Window

Top panel

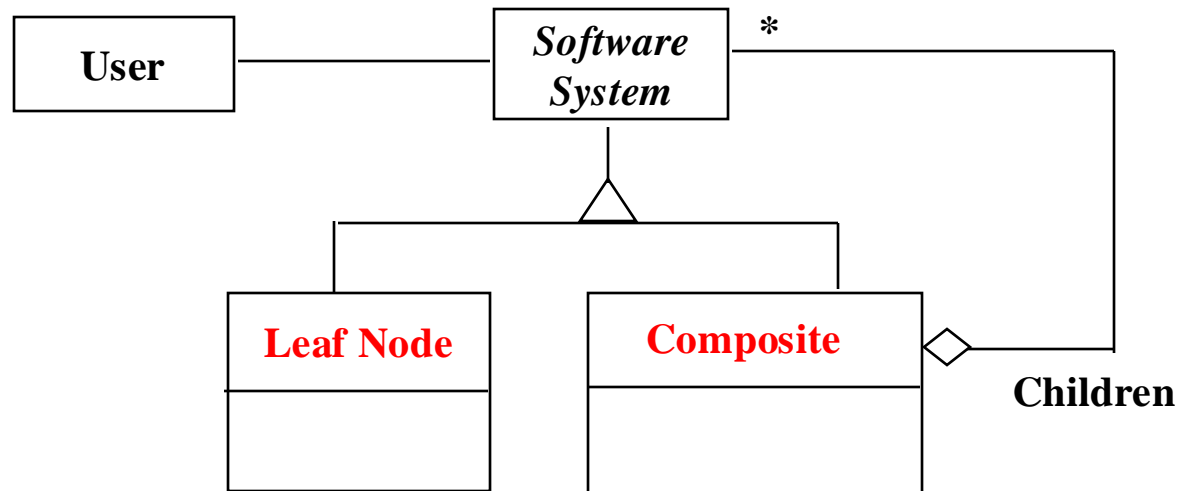
Main panel

Button panel



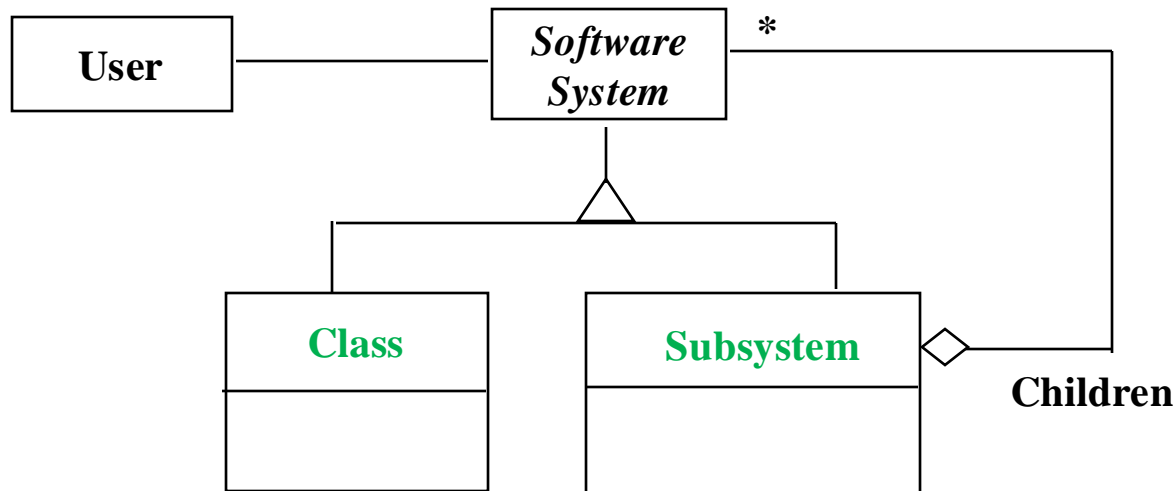
Composite Pattern: esercizio

- **Vogliamo modellare un sistema software con il pattern composite:**
 - **Definizione:** A software system consists of subsystems which are either other Subsystems or collection of Classes
 - **Leaf node:** ?
 - **Composite:** ?

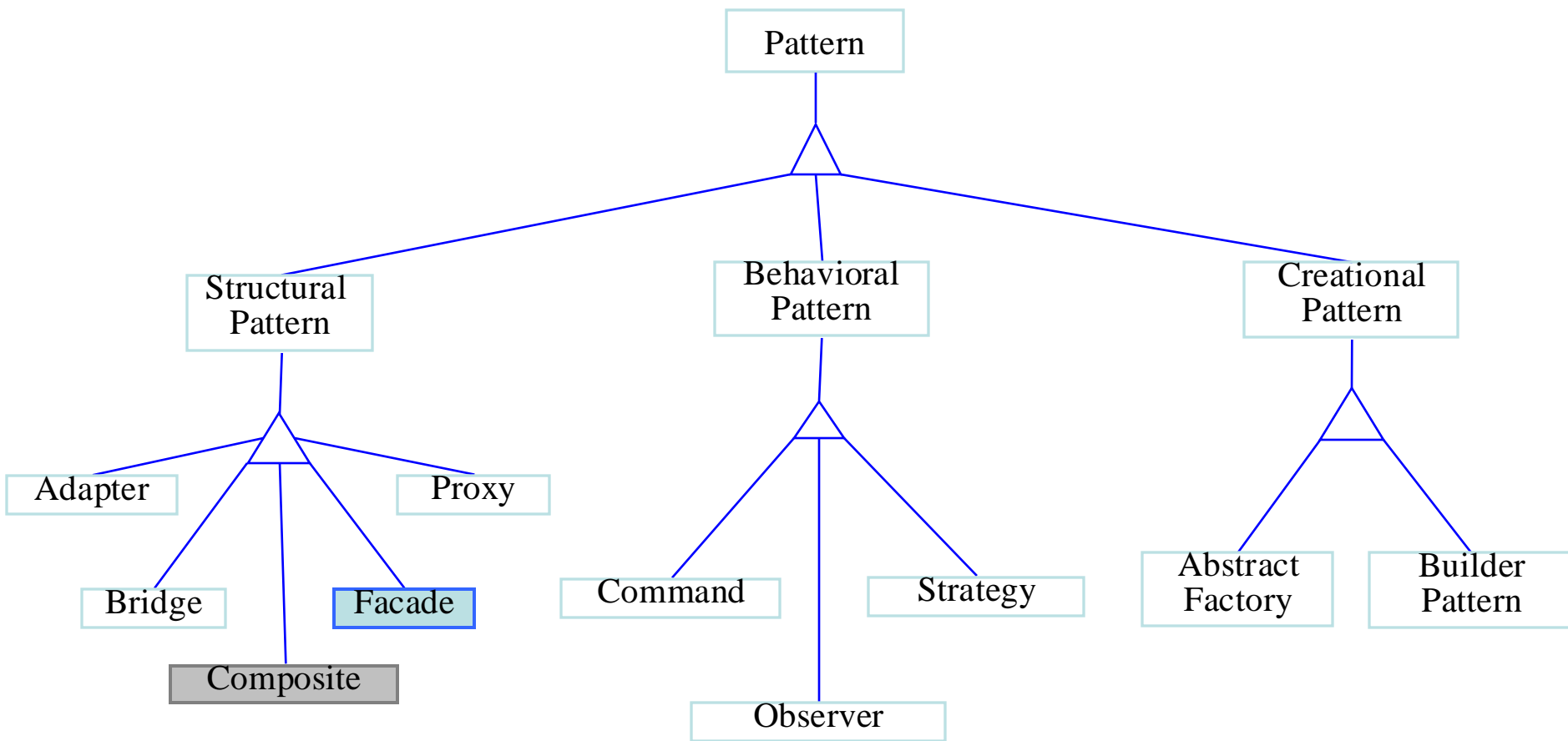


Composite Pattern: soluzione

- **Vogliamo modellare un sistema software con il pattern composite:**
 - **Definizione:** A software system consists of subsystems which are either other Subsystems or collection of Classes
 - **Leaf node:** Class
 - **Composite:** Subsystem

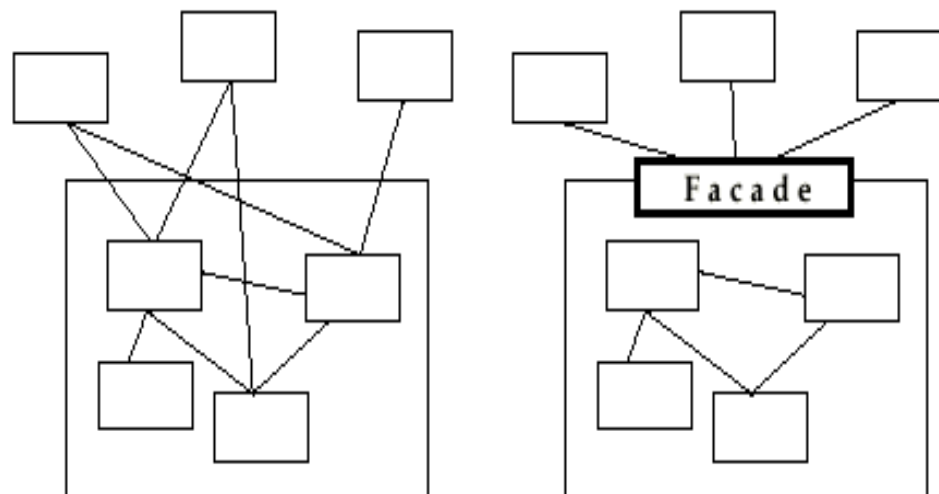


Pattern strutturali: Facade Pattern



Facade pattern

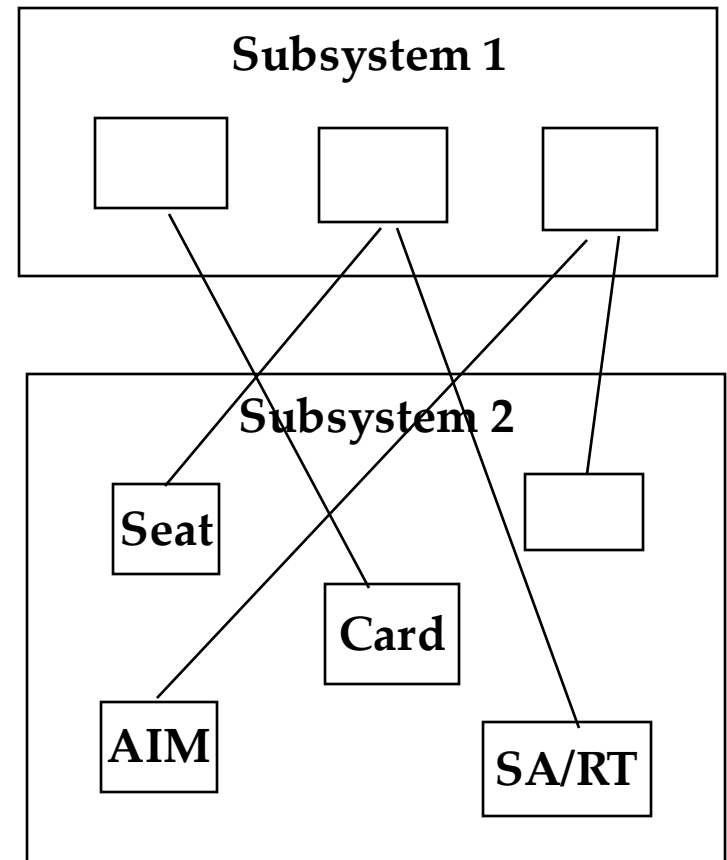
- **Nome:** Facade
- **Descrizione del problema:**
 - Fornisce un'unica interfaccia per accedere ad un insieme di oggetti che compongono un sottosistema.
- **Possibili applicazioni:**
 - A facade pattern should be used by all subsystems in a software system.
 - The façade defines all the services of the subsystem.
 - Facades allow us to provide a closed architecture



Facade Pattern: esempio (1)

- Subsystem 1 can look into the Subsystem 2 (vehicle subsystem) and call on any component or class operation at will.
- Why is this good?
 - Efficiency
- Why is this bad?
 - Can't expect the caller to understand how the subsystem works or the complex relationships within the subsystem.
 - We can be assured that the subsystem will be misused, leading to non-portable code.

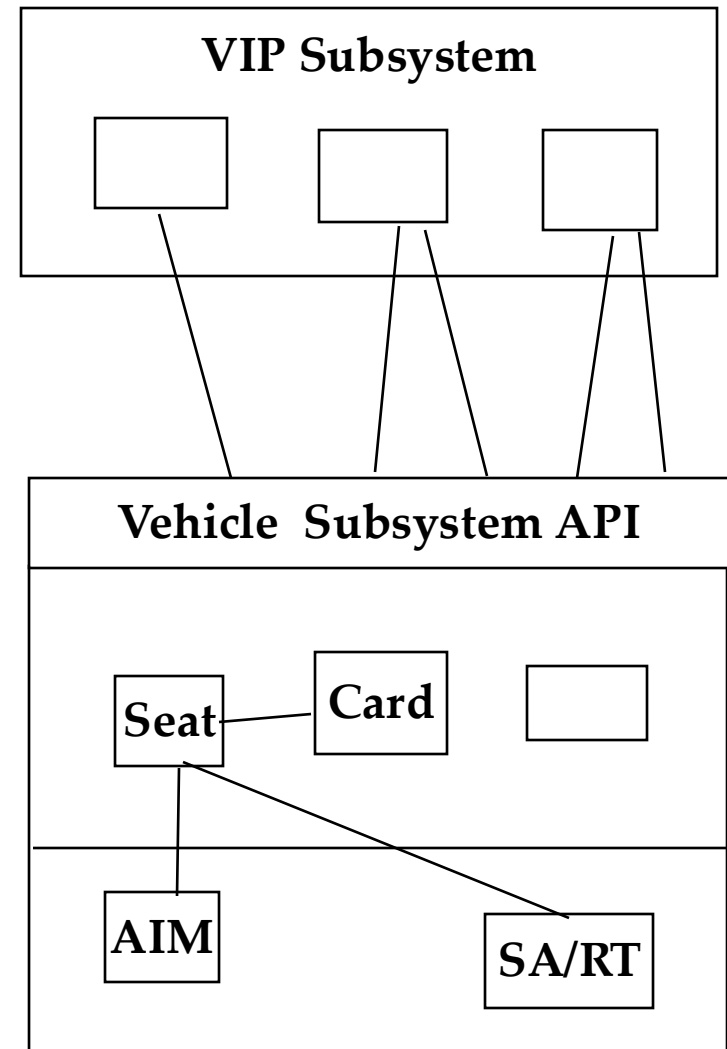
Architettura aperta



Facade Pattern: esempio (2)

- The subsystem decides exactly how it is accessed.
- No need to worry about misuse by callers.
- If a façade is used the subsystem can be used in an early integration test.
 - We need to write only a driver.

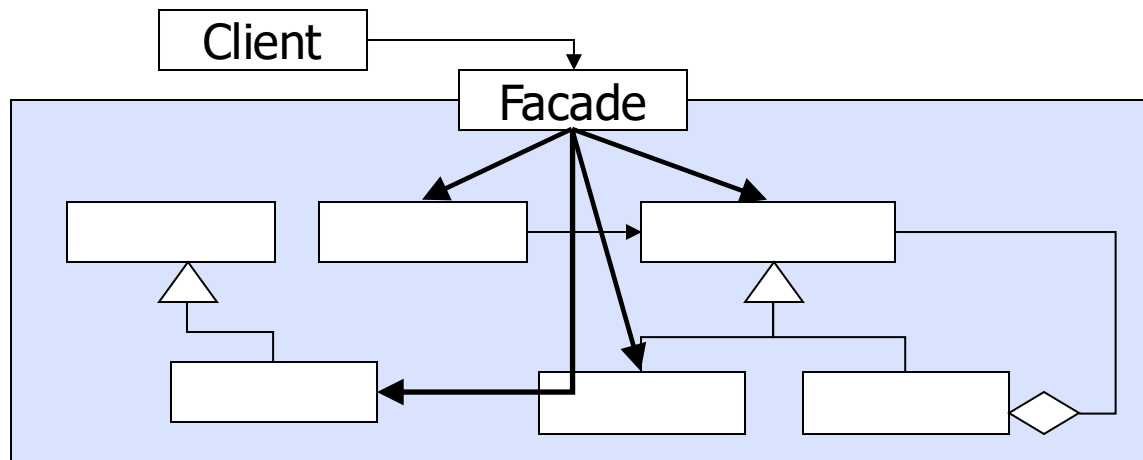
Architettura chiusa



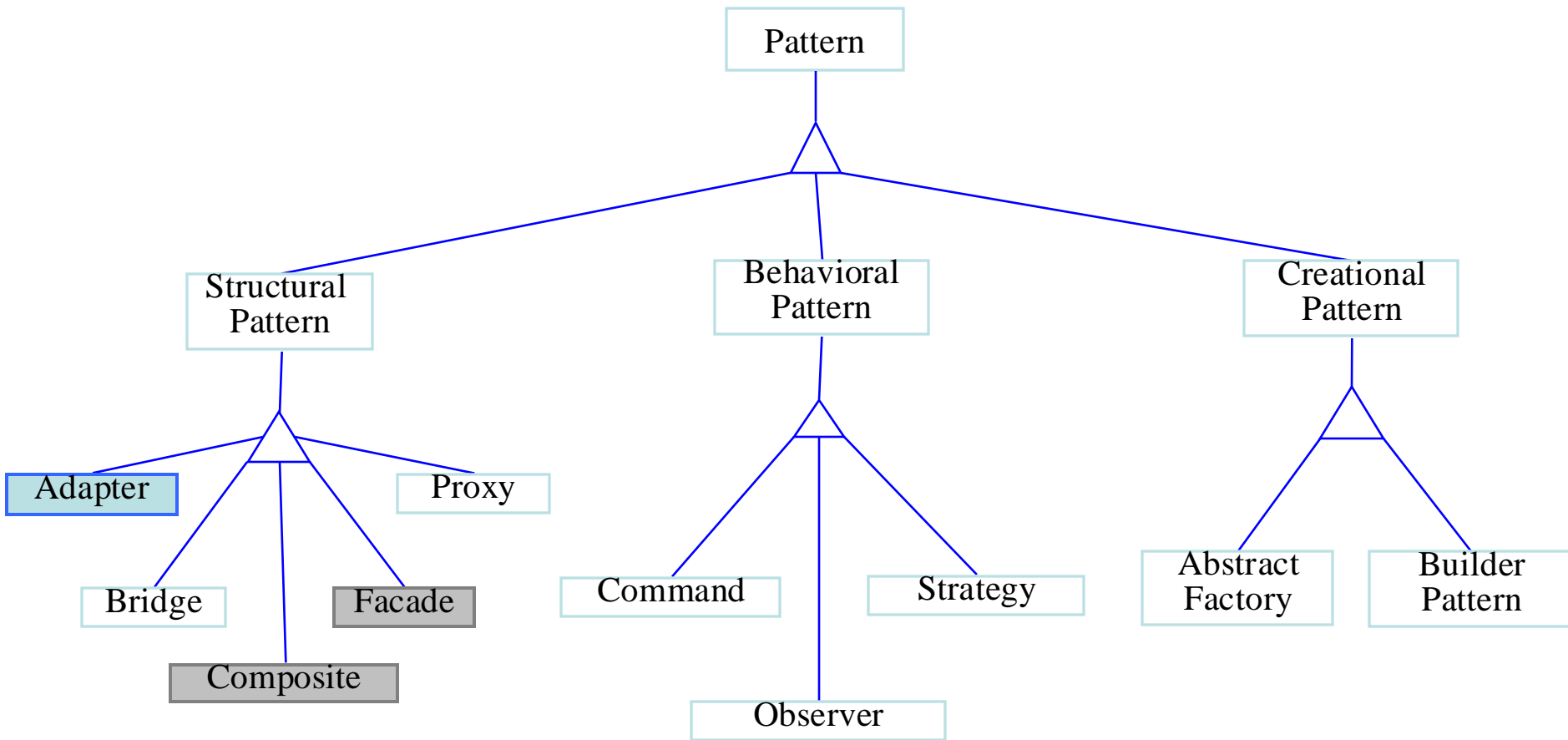
Facade Pattern

- **Conseguenze:**

- Provide **unified interface** to interfaces **within a subsystem**.
- Define a **higher-level interface** that makes the **subsystem easier to use** (i.e. it abstracts out the subsystem details).
- **Shield clients from subsystem components**.
- Promote **weak coupling** between client and subsystem components.



Pattern strutturali: Adapter Pattern



Incapsulare componenti con il Pattern Adapter

- Quando la complessità di un sistema aumenta ed il tempo per rilasciare il prodotto diminuisce, il costo dello sviluppo del software eccede di molto i costi dell' hardware.
- Quindi gli sviluppatori sono incentivati a riutilizzare componenti da progetti precedenti o ad usare componenti off-the-shelf (COTS).
 - Le componenti esistenti sono incapsulate.
 - In questo modo si ha una separazione tra il sistema e le componenti minimizzando l' impatto dei nuovi software sul nuovo progetto.
- Questo problema può essere risolto utilizzando il pattern *Adapter*.

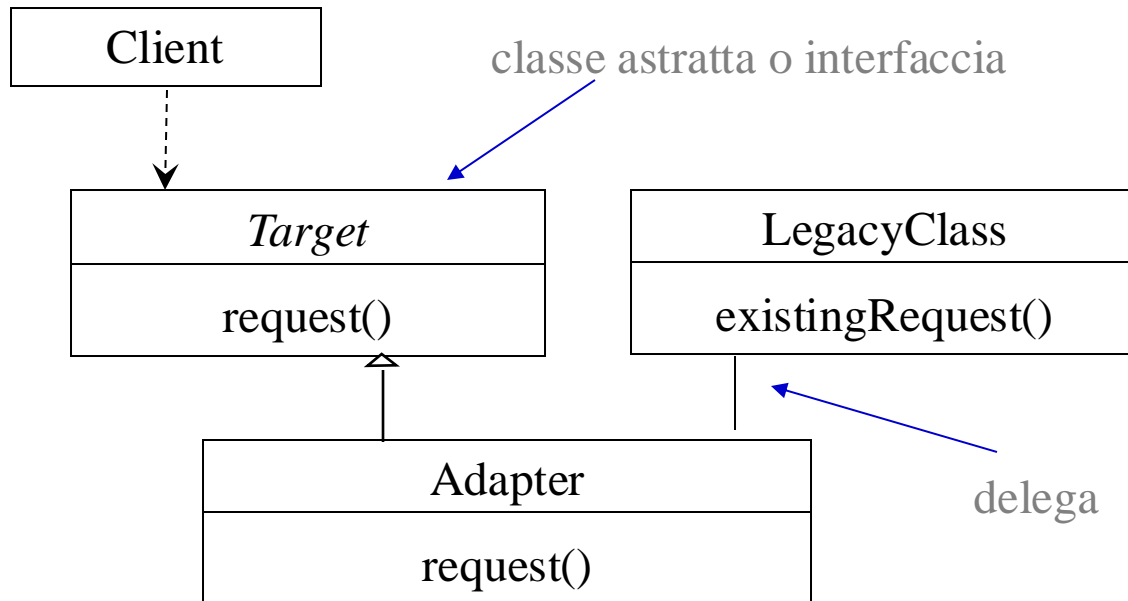
Adapter Pattern (1)

- **Nome:** Adapter (Wrapper)
- **Descrizione del problema:**
 - Convertire l'interfaccia di una classe in una interfaccia diversa che il cliente si aspetta, in maniera tale che classi diverse possano operare insieme nonostante abbiamo interfacce incompatibili.
- **Possibili applicazioni:**
 - Incapsulare componenti esistenti per riutilizzare componenti da progetti precedenti o usare componenti off-the-shelf (COTS).
 - Fornire una nuova interfaccia a componenti legacy esistenti (interface engineering, reengineering).
 - Esempio: *Come utilizzare un Web Browser per accedere ad un sistema di information retrieval esistente?*

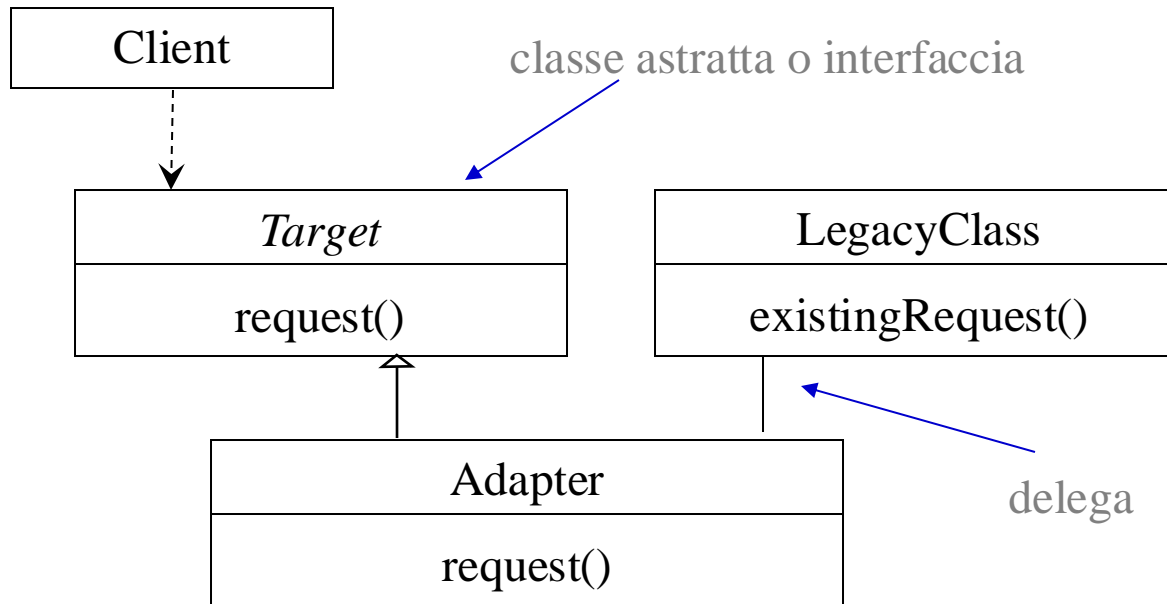
Adapter Pattern (2)

- **Soluzione:**

- Ogni metodo dell'interfaccia *Target* è implementato in termini di richieste alla classe *LegacyClass*.
- Ogni conversione tra strutture dati o variazioni del comportamento sono realizzate dalla classe *Adapter*.



Adapter Pattern (3)

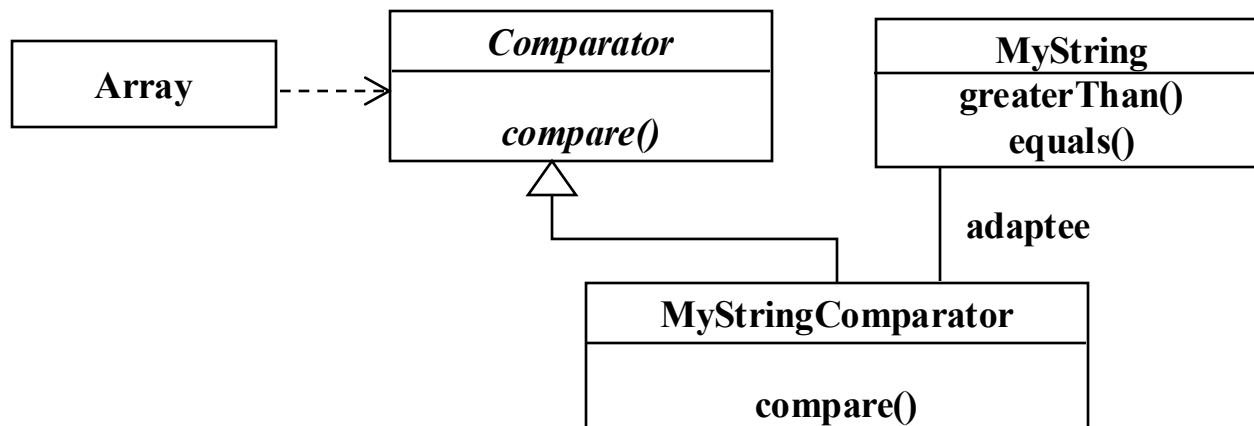


- **Conseguenze:**

- Se *Client* utilizza *Target* allora può utilizzare qualsiasi istanza dell'*Adapter* in maniera trasparente senza dover essere modificato.
- *Adapter* lavora con la classe *LegacyClass* e con tutte le sue sottoclassi.
- L' *Adapter* pattern viene utilizzato quando l'interfaccia (es: *Target*) e la sua implementazione (es: *LegacyClass*) esistono già e non possono essere modificate.

Adapter Pattern: esempio (1)

- Assumiamo che il **codice client** è il metodo `sort(Object[] a, Comparator c)` statico della classe Java **Array**
 - L'interfaccia **Comparator** fornisce un metodo `compare()` per definire l'ordine relativo tra gli elementi contenuti nell'array *a* di *Object*.
- Assumiamo di voler **ordinare** l'array di oggetti **MyString**
 - MyString** offre i metodi `greaterThan()` e `equals()` per il confronto
 - dobbiamo **definire un nuovo comparatore** **MyStringComparator** che fornisce il metodo `compare()` usando `greaterThan()` e `equals()`



Adapter Pattern: esempio (2)

// Existing Client

```
class Array {  
    static void sort(Object [] a,  
        Comparator c);  
    /* ... */  
}
```

// Existing target interface

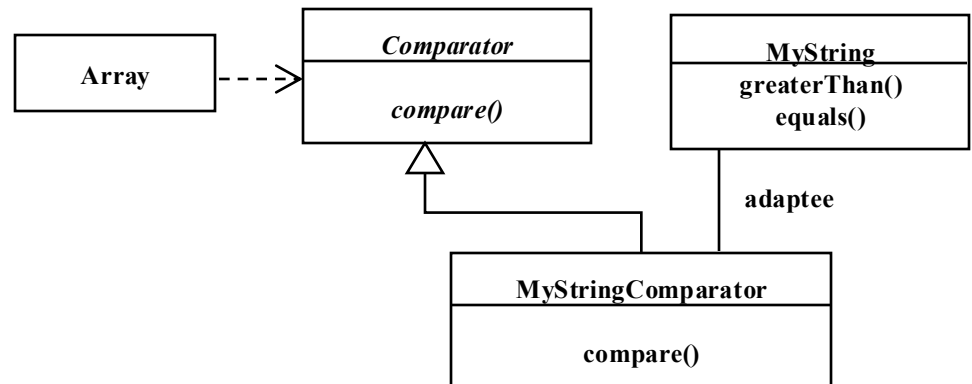
```
interface Comparator {  
    int compare(Object o1, Object o2);  
    /* ... */  
}
```

// Existing adaptee class

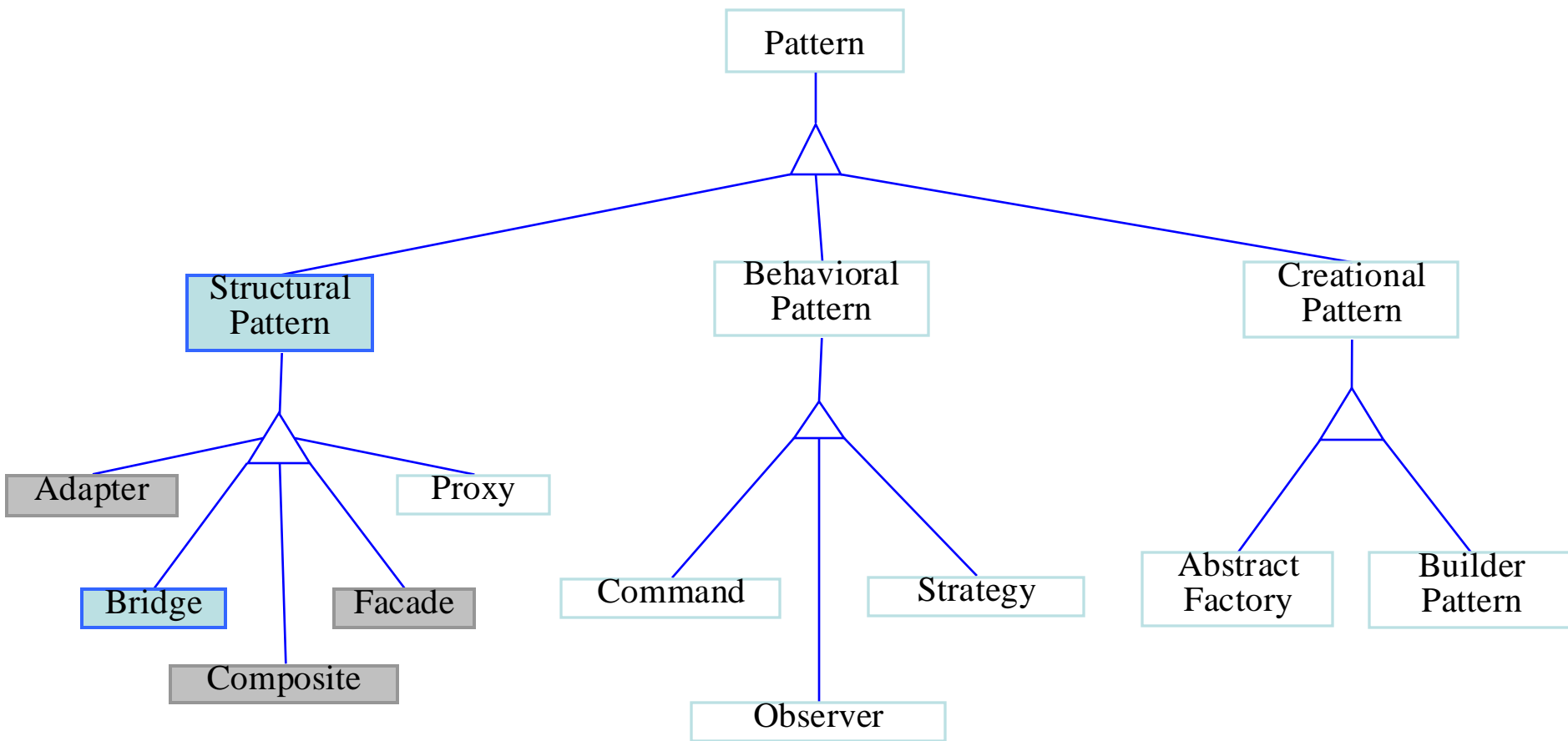
```
class MyString extends String {  
    boolean equals(Object o);  
    boolean greaterThan(MyString s);  
    /* ... */  
}
```

// Adapter class

```
class MyStringComparator implements  
    Comparator {  
    /* ... */  
    int compare(Object o1, Object o2) {  
        int result;  
        if (o1.greaterThan(o2) {  
            result=1;  
        } else if (o1.equals(o2) {  
            result=0;  
        } else {  
            result=-1  
        }  
        return result;  
    }  
}
```



Pattern strutturali: Bridge Pattern



Integrare sottosistemi con il Bridge Pattern

- Consideriamo il problema di sviluppare, testare ed integrare sottosistemi realizzati da differenti sviluppatori in maniera incrementale.
- Per evitare che l'integrazione dei sottosistemi venga ritardata fin quando tutti i sottosistemi siano stati realizzati:
 - si utilizzano implementazioni di stub (invece di uno specifico sottosistema);
 - si sviluppano diverse implementazioni dello stesso sottosistema.
- È necessaria una soluzione che consenta di sostituire dinamicamente realizzazioni diverse della stessa interfaccia per usi differenti.
- Questo problema può essere risolto utilizzando il design pattern *Bridge*.

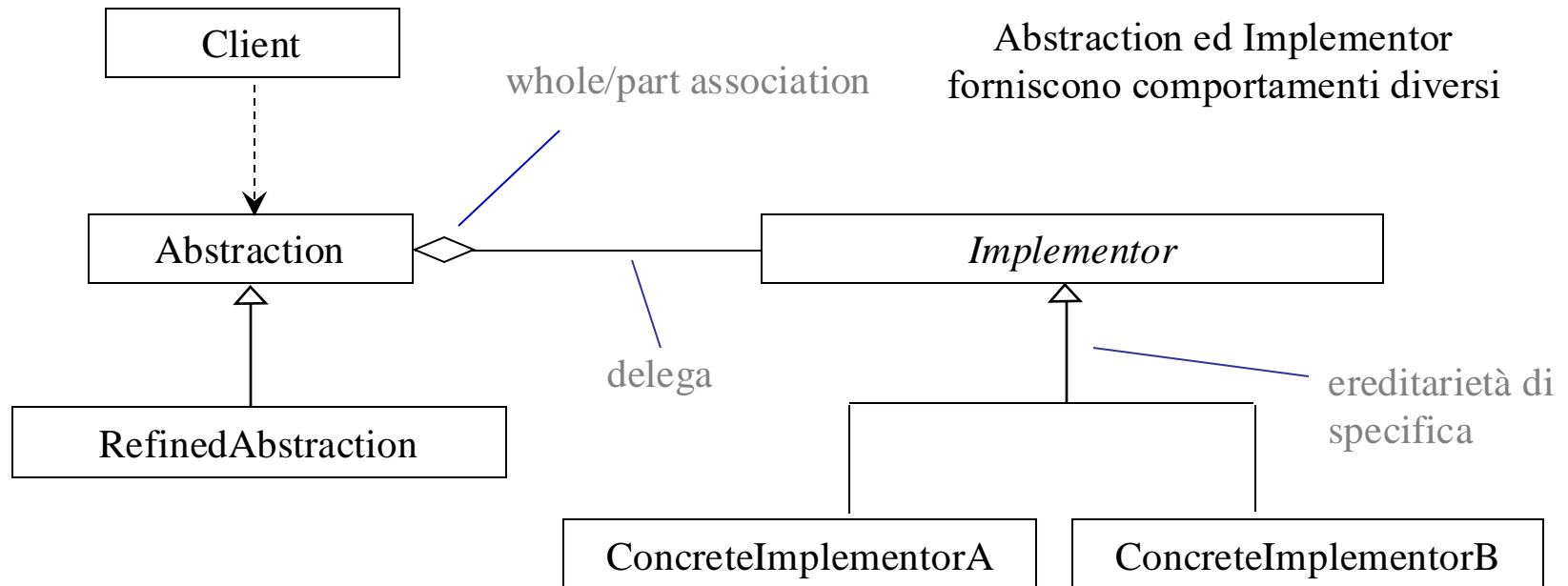
Bridge Pattern (1)

- **Nome:** Bridge (Handle/Body)
- **Descrizione del problema:**
 - Separare un'astrazione da una implementazione così che una diversa implementazione possa essere sostituita, eventualmente a runtime (es. testare differenti implementazioni della stessa interfaccia).
- **Possibili applicazioni:**
 - Utile per interfacciare un insieme di oggetti:
 - quando l'insieme non è ancora completamente noto (ad es. in fase di analisi, design, testing).
 - quando c'è necessità di estendere un sottosistema dopo che il sistema è stato consegnato ed è in esecuzione (estensione dinamica).

Bridge Pattern (2)

- **Soluzione:**

- Una classe *Abstraction* definisce l'interfaccia visibile al codice *Client*.
- *Implementor* è una interfaccia astratta che definisce i metodi di basso livello disponibili ad *Abstraction*.
- Un'istanza di *Abstraction* mantiene un riferimento alla corrispondente istanza di *Implementor*
- *Abstraction* e *Implementor* possono essere raffinate indipendentemente



Bridge Pattern: esempio

```
// Client
Event e = new Event();
AbstractEventEngine aee = new
    RefinedAbstractEventEngine();
EventEngine ee = aee.getEventEngineImplementor();
ee.notify(e);
```

```
// Implementor
public interface EventEngine{
    public void notify(Event e);
}
```

```
// Abstraction
public abstract class AbstractEventEngine {
    protected EventEngine imp;

    public abstract EventEngine
        getEventEngineImplementor();
}
```

```
// Concrete Implementor
public class ConcreteEventEngine implements
    EventEngine {
    public void notify(Event e) {
        /* Implementazione concreta */
    }
}
```

```
// RefinedAbstraction
public class RefinedAbstractEventEngine extends
    AbstractEventEngine{

    public EventEngine getEventEngineImplementor(){
        //imp = new EventEngineStub();
        imp = new ConcreteEventEngine();
        return imp;
    }
}
```

```
//ConcreteImplementor (Stub)
public class EventEngineStub implements
    EventEngine {
    public void notify(Event e){
        System.out.println(
            "Ho inviato l'evento " + e);
    }
}
```

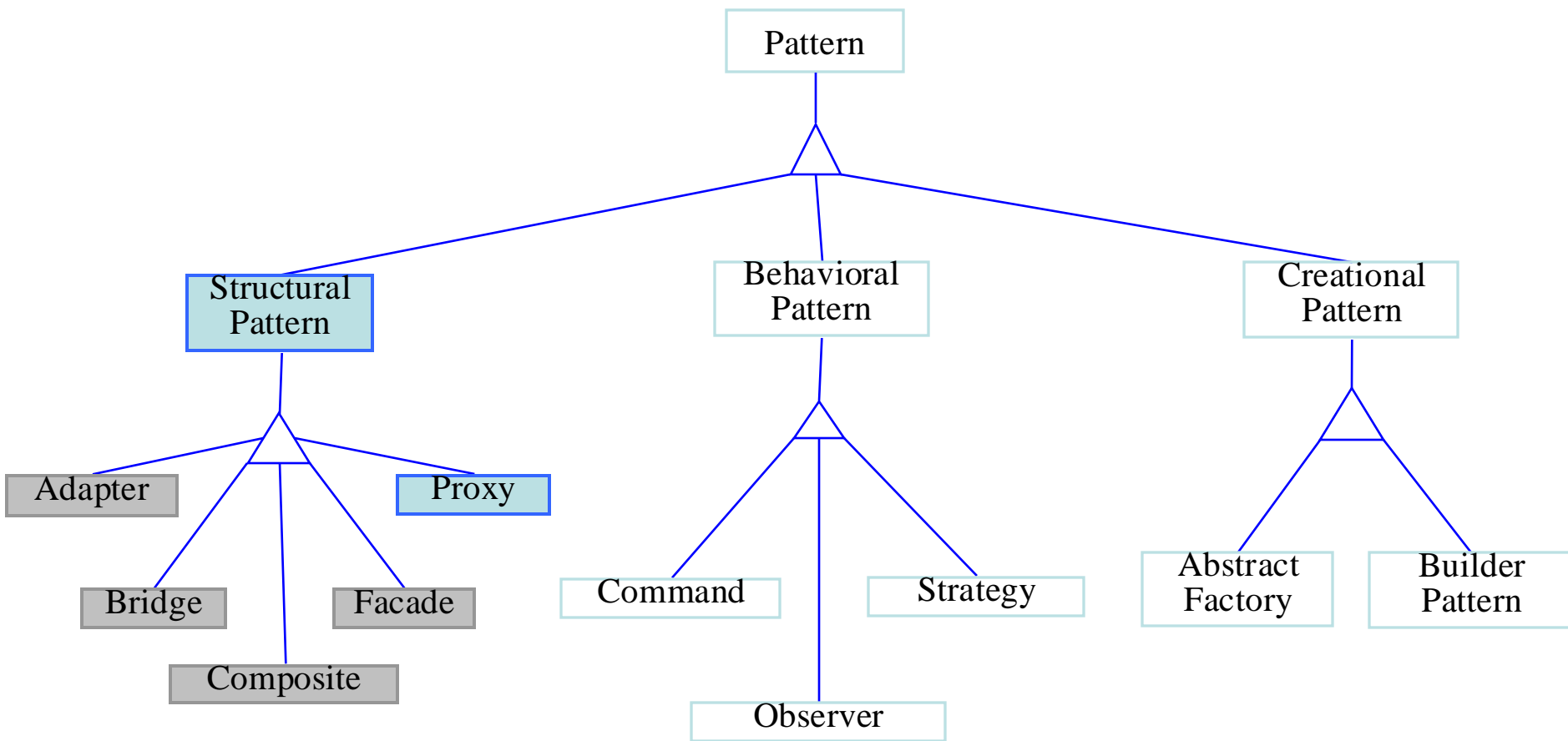
Bridge Pattern: conseguenze

- **Conseguenze:**
 - *Disaccoppiamento tra interfaccia ed implementazione.*
 - Un'implementazione non è più legata in modo permanente ad un'interfaccia.
 - L'implementazione di un'astrazione può essere configurata durante l'esecuzione.
 - La parte di alto livello di un sistema dovrà conoscere soltanto le classi Abstraction e Implementor.
 - *Maggiore estendibilità*
 - Le gerarchie Abstraction e Implementor possono essere estese indipendentemente.
 - *Mascheramento dei dettagli dell'implementazione ai client*
 - I client non devono preoccuparsi dei dettagli implementativi.

Adapter Pattern VS Bridge Pattern

- Similarities:
 - Both are used to hide the details of the underlying implementation.
- Difference:
 - The adapter pattern is geared towards making unrelated components work together
 - Applied to systems after they're designed (reengineering, interface engineering).
 - A bridge, on the other hand, is used up-front in a design to let abstractions and implementations vary independently.
 - Green-field engineering of an “extensible system”.
 - New “beasts” can be added to the “object zoo”, even if these are not known at analysis or system design time.

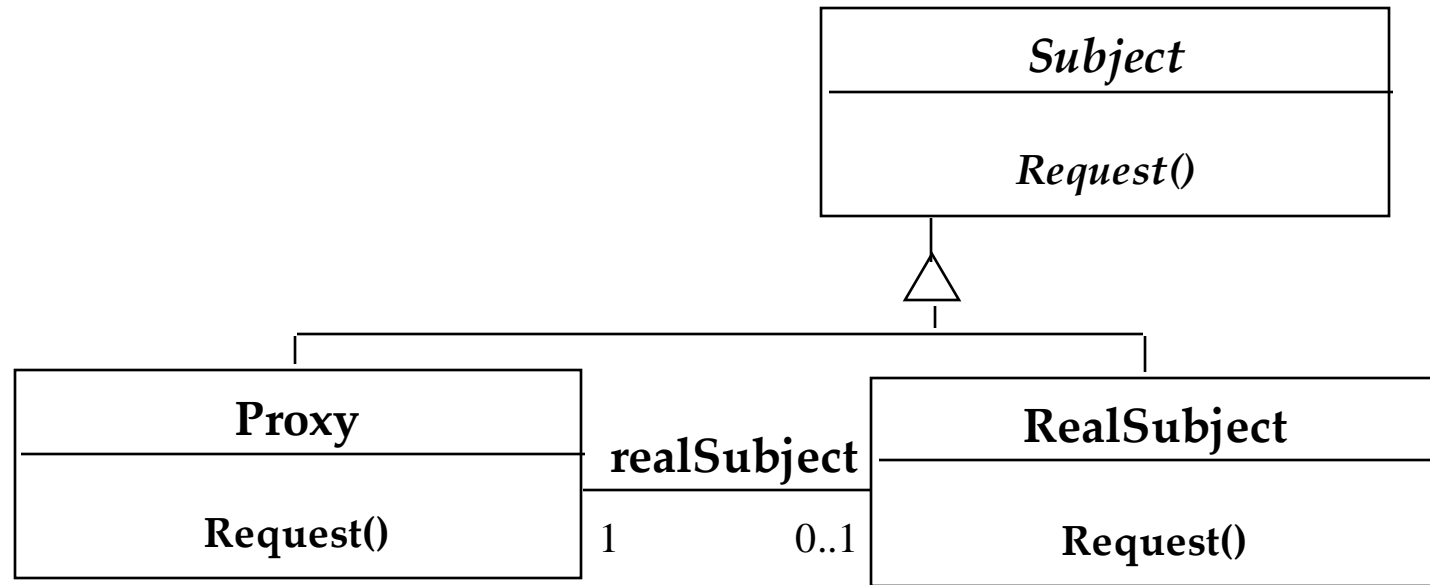
Pattern strutturali: Proxy Pattern



Proxy Pattern (1)

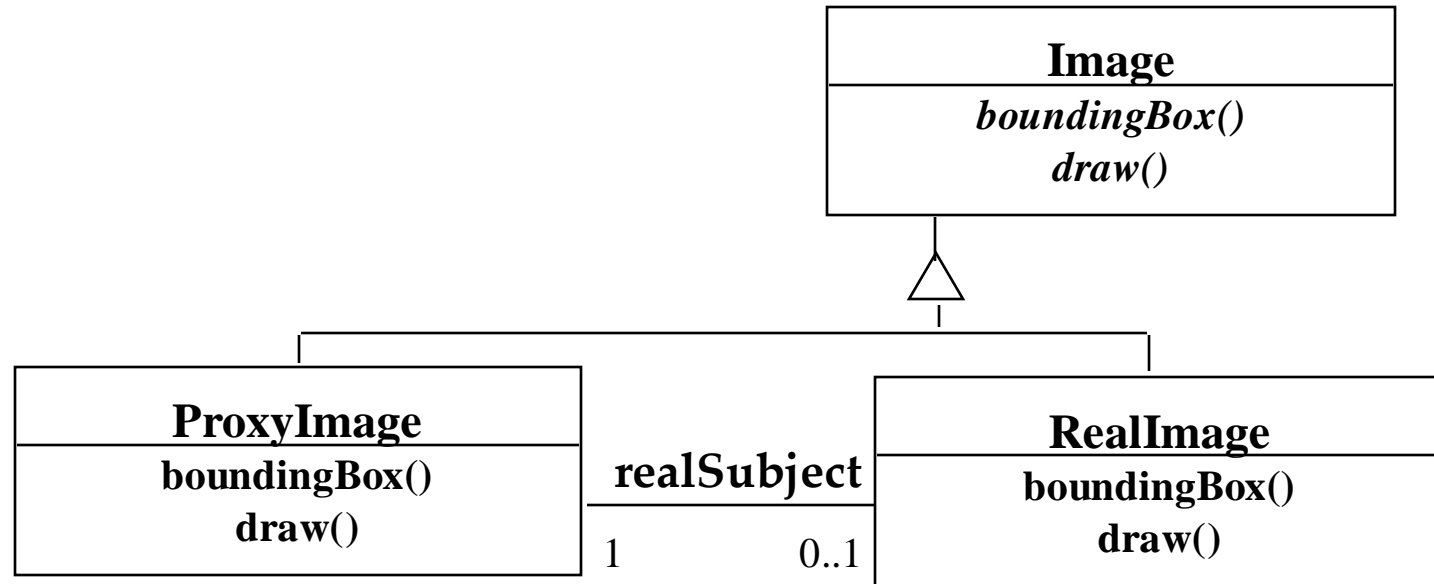
- What is expensive?
 - Object Creation.
 - Object Initialization.
- Defer object creation and object initialization to the time you need the object.
- Proxy pattern:
 - Reduces the cost of accessing objects.
 - Uses another object (“the proxy”) that acts as a stand-in for the real object.
 - The proxy creates the real object only if the user asks for it.

Proxy Pattern (2)



- *Interface inheritance* is used to specify the interface shared by **Proxy** and **RealSubject**.
- *Delegation* is used to catch and forward any accesses to the **RealSubject** (if desired)
- Proxy patterns can be used for lazy evaluation and for remote invocation.
- Proxy patterns can be implemented with a Java interface.

Virtual Proxy example

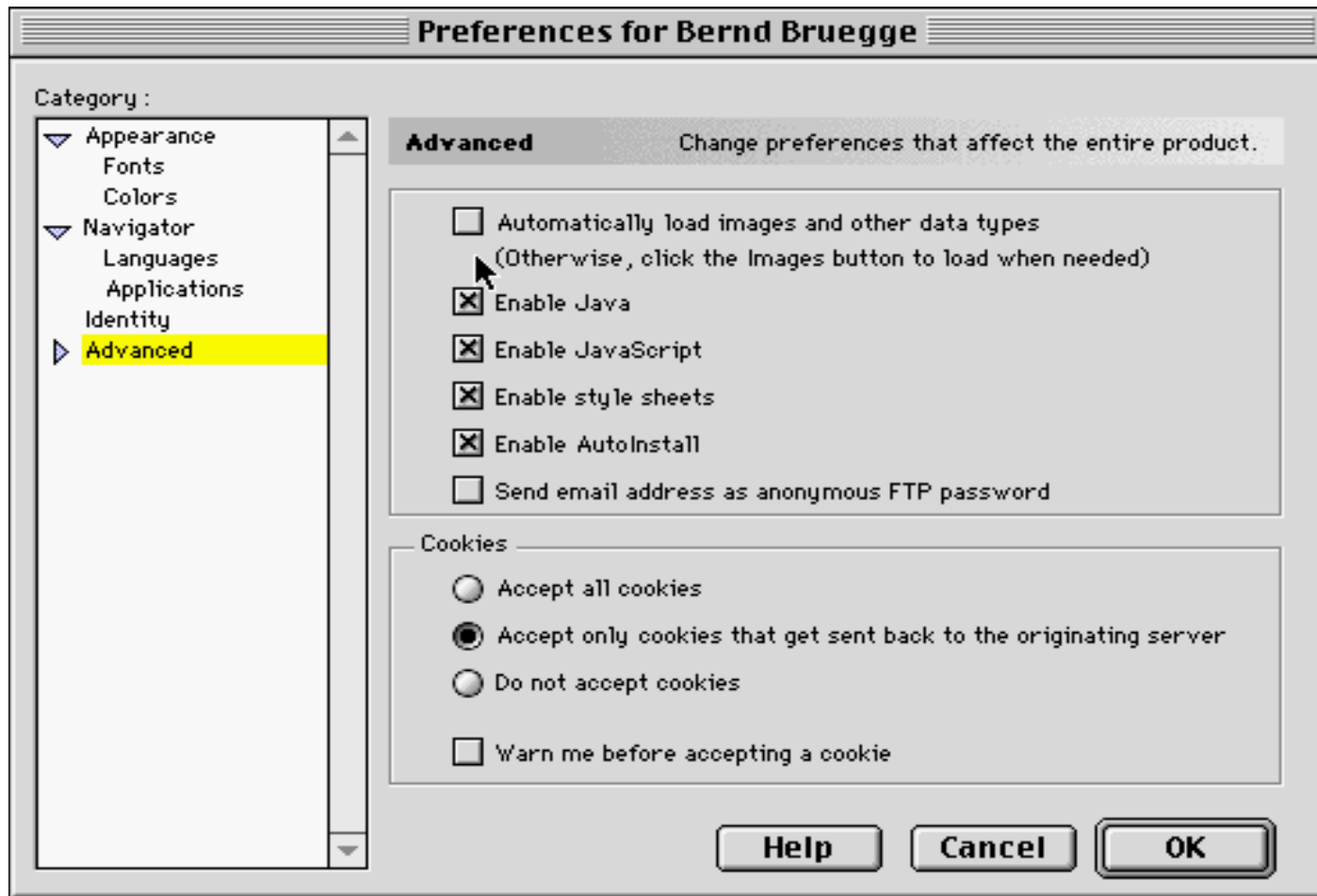


- **Images** are stored and loaded separately from text.
- If a **RealImage** is not loaded a **ProxyImage** displays a grey rectangle in place of the image.
- The client cannot tell that it is dealing with a **ProxyImage** instead of a **RealImage**.
- A proxy pattern can be easily combined with a **Bridge**.

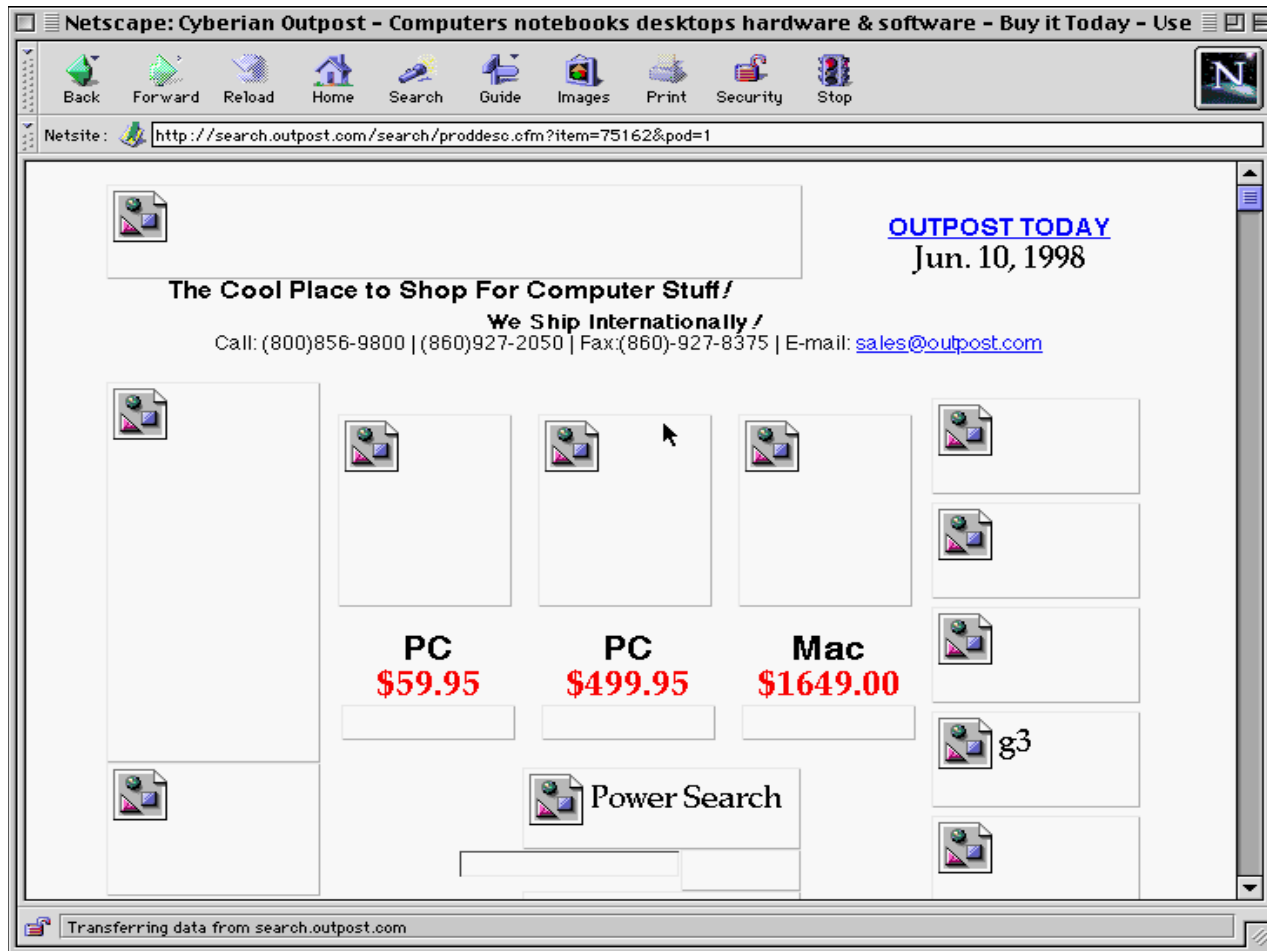
Before



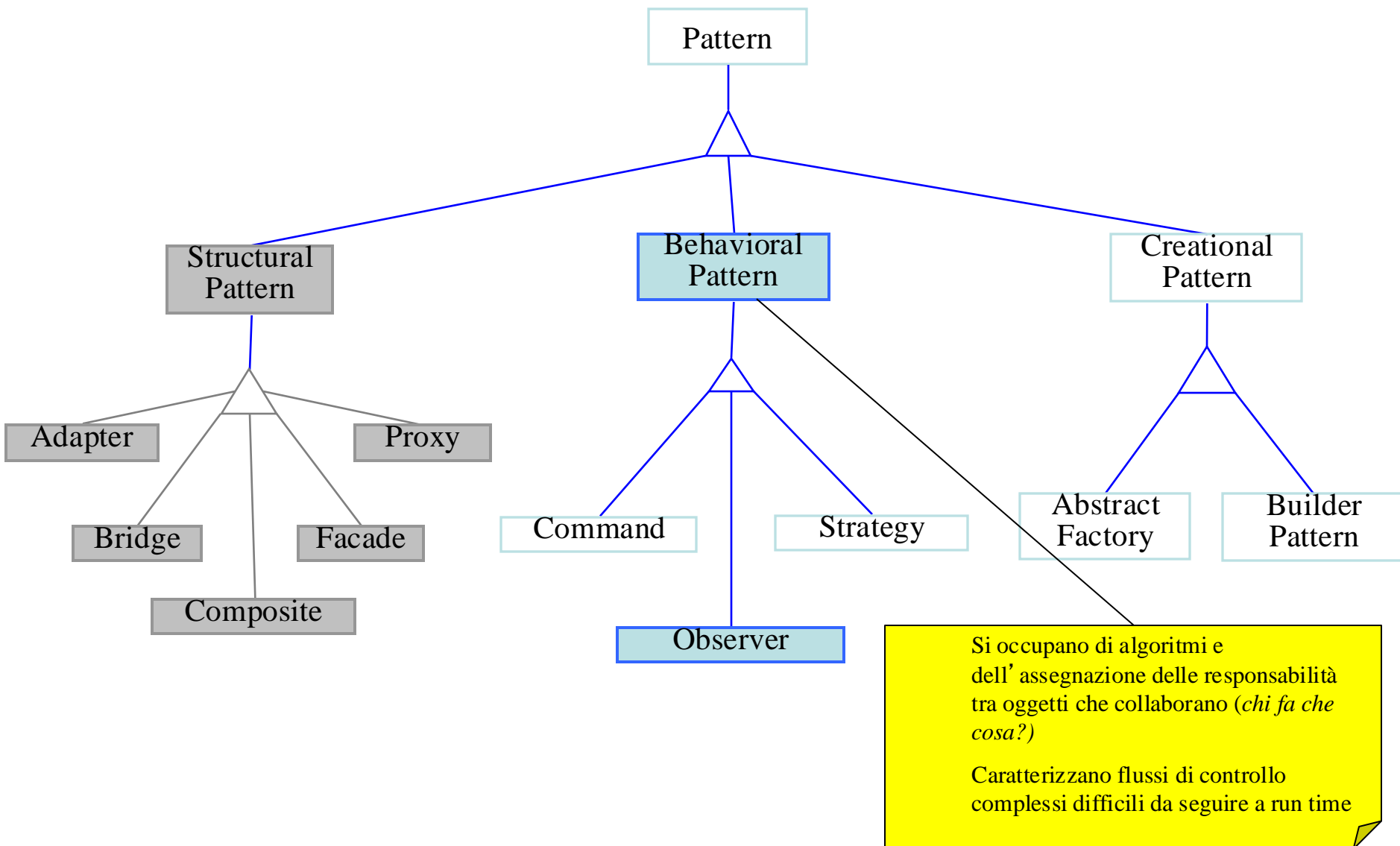
Controlling Access



After



Pattern comportamentali: Observer Pattern



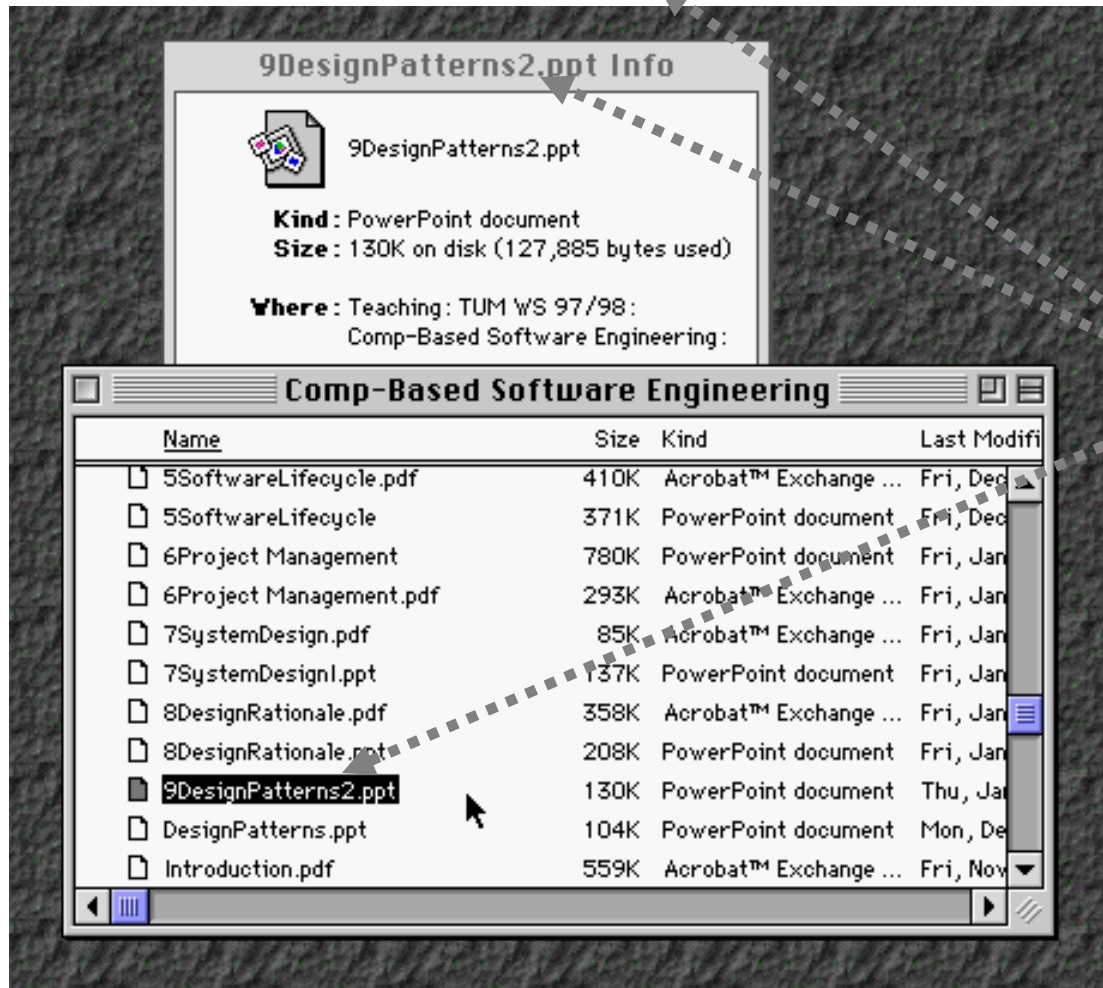
Observer Pattern (1)

- “Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.”
- Also called “Publish and Subscribe”.
- Uses:
 - Maintaining consistency across redundant state.
 - Optimizing batch changes to maintain consistency.

Observer Pattern (2)

Observers

Subject



9DesignPatterns2.ppt

Observer Pattern (2)

- La figura a destra rappresenta il diagramma delle classi che lega la classe JButton all'interfaccia ActionListener
- Come si vede, c'è una corrispondenza diretta con lo schema proprio del pattern OBSERVER, riportato sulla slide precedente
- In particolare, JButton ricopre il ruolo di Subject e ActionListener quello di Observer

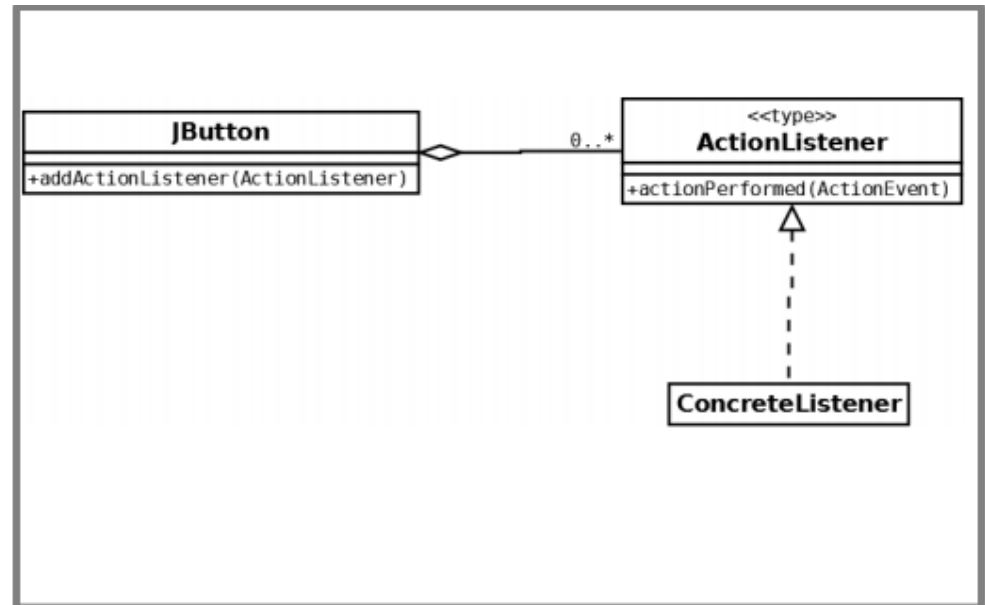
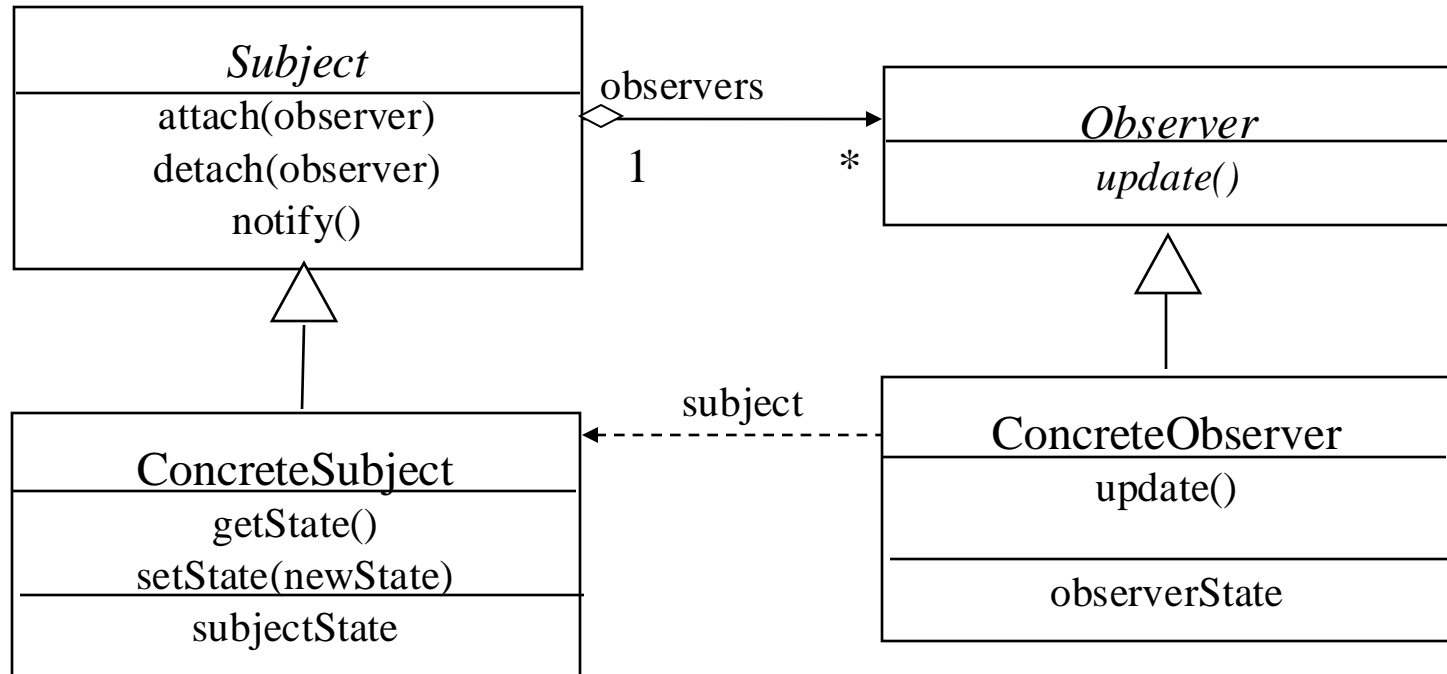


Figura 4: Diagramma UML dell'applicazione del pattern OBSERVER ai pulsanti delle interfacce grafiche create con Swing/AWT.

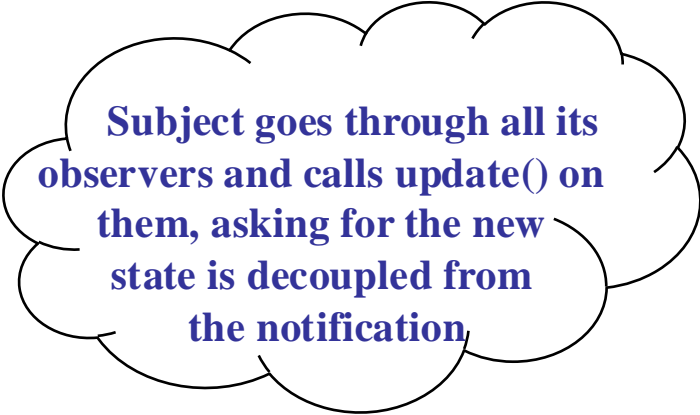
Observer Pattern (3)



- The **Subject** represents the actual state, the **Observers** represent different views of the state.
- **Observer** can be implemented as a Java interface.
- **Subject** is a super class (needs to store the observers vector) *not* an interface.

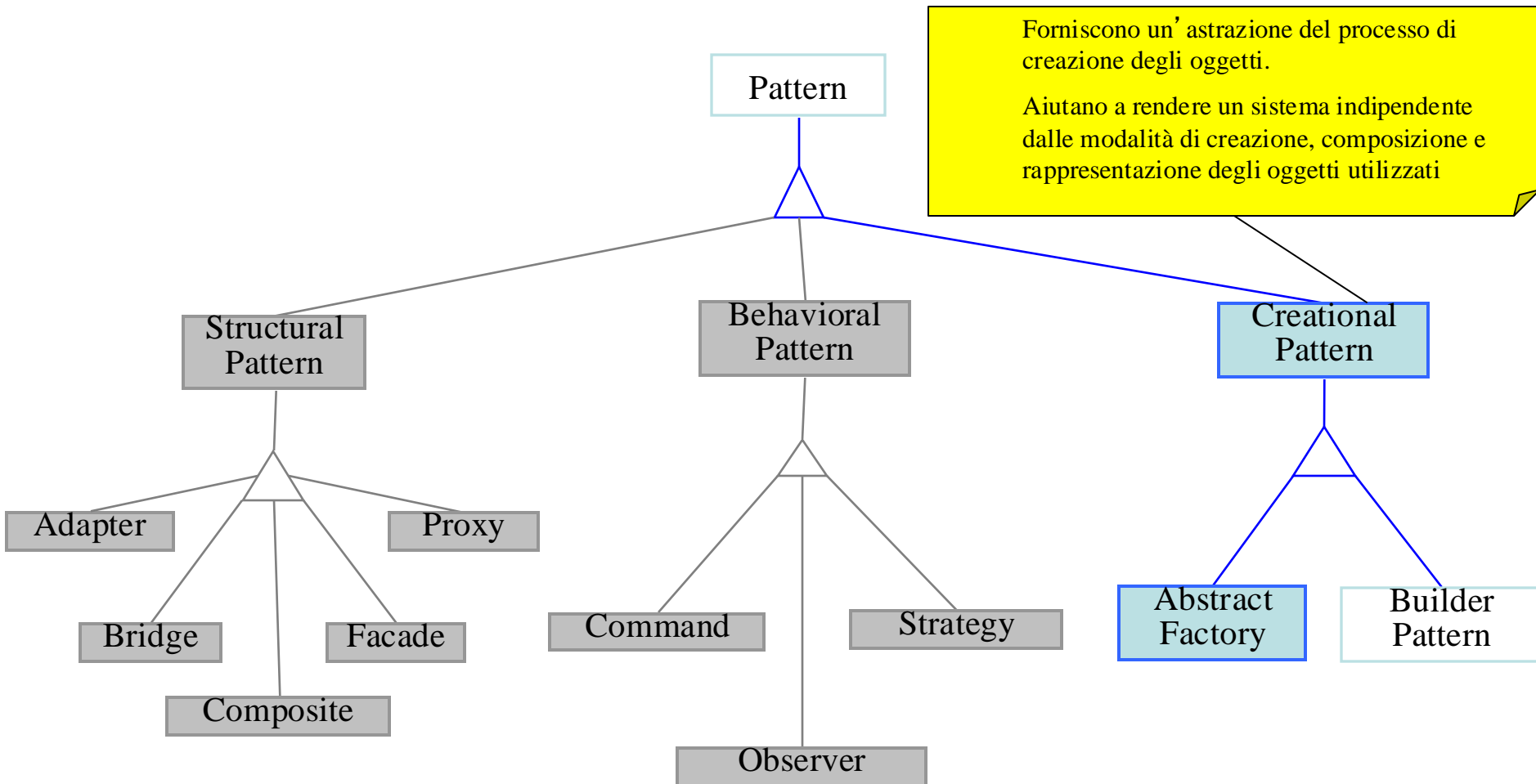
Implementazione Java del Pattern Observer

```
import java.util;  
  
public class Observable extends Object {  
    public void addObserver(Observer o);  
    public void deleteObserver(Observer o);  
    public boolean hasChanged();  
    public void notifyObservers();  
    public void notifyObservers(Object arg);  
}  
  
public class Subject extends Observable {  
    public void setState(String filename);  
    public String getState();  
}  
  
public abstract interface Observer {  
    public abstract void update(Observable o, Object arg);  
}
```



**Subject goes through all its
observers and calls update() on
them, asking for the new
state is decoupled from
the notification**

Pattern Creazionali: AbstractFactory Pattern



Incapsulare le piattaforme con il Pattern AbstractFactory

- Consideriamo un' applicazione per un' abitazione intelligente
 - l' applicazione riceve eventi da sensori ed attiva i comandi per diversi dispositivi.
- L' interoperabilità in questo dominio è scarsa, di conseguenza è difficile sviluppare un singolo sistema software di controllo che funzioni per dispositivi prodotti da diverse aziende.
- Questo problema può essere risolto utilizzando il pattern *Abstract Factory*.

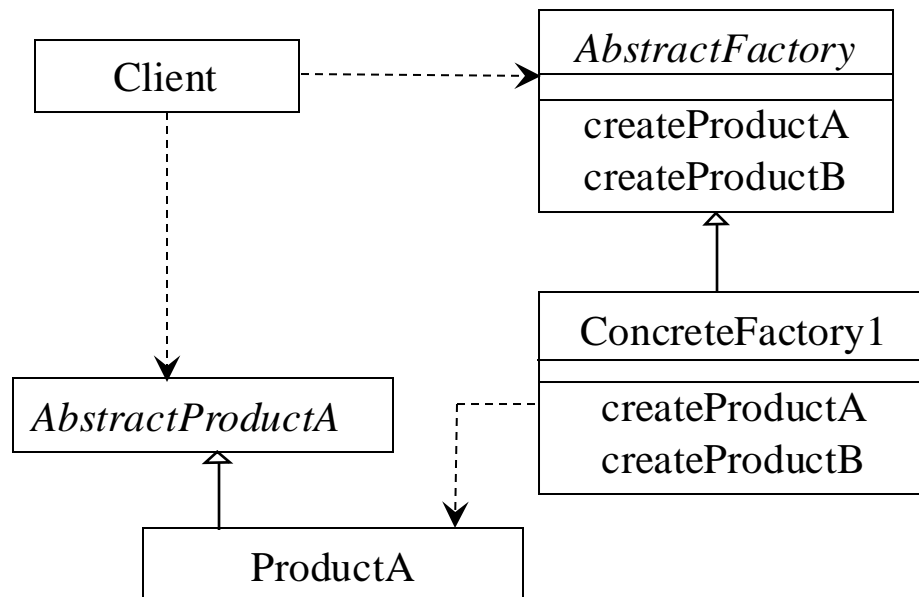
Abstract Factory Pattern (1)

- **Nome:** Abstract Factory
- **Descrizione del problema:**
 - Fornire un'interfaccia per la creazione di famiglie di oggetti correlati o dipendenti senza specificare quali siano le loro classi concrete.
- **Soluzione:**
 - Una piattaforma (es., un sistema per la gestione di finestre) è rappresentato con un insieme di ***AbstractProducts***, ciascuno dei quali rappresenta un concetto (es., un bottone).
 - Una classe ***AbstractFactory*** dichiara le operazioni per creare ogni singolo prodotto.
 - Una piattaforma specifica è poi realizzata da un ***ConcreteFactory*** ed un insieme di ***ConcreteProducts***.

Abstract Factory Pattern (2)

- **Soluzione:**

- Una piattaforma (es., un sistema per la gestione di finestre) è rappresentato con un insieme di **AbstractProducts**, ciascuno dei quali rappresenta un concetto (es., un bottone). Una classe **AbstractFactory** dichiara le operazioni per creare ogni singolo prodotto.
- Una piattaforma specifica è poi realizzata da un **ConcreteFactory** ed un insieme di **ConcreteProducts**.



Le Classi per ProductB non sono illustrate.

Ci possono essere diverse classi ConcreteFactory, ciascuna delle quali è una sottoclasse di AbstractFactory

Abstract Factory Pattern (3)

- **Conseguenze:**

- *Isola le classi concrete*

- Abstract Factory incapsula la responsabilità e il processo di creazione di oggetti prodotto e rende il client indipendente dalle classi effettivamente utilizzate per l'implementazione degli oggetti.

- *Permette di sostituire facilmente famiglie di prodotti a runtime*

- una factory concreta compare solo quando deve essere istanziata.

- *Promuove la coerenza nell'utilizzo dei prodotti*

- famiglia di prodotto correlati è solitamente progettata per essere utilizzata insieme
 - il Client può creare prodotti solo utilizzando la AbstractFactory

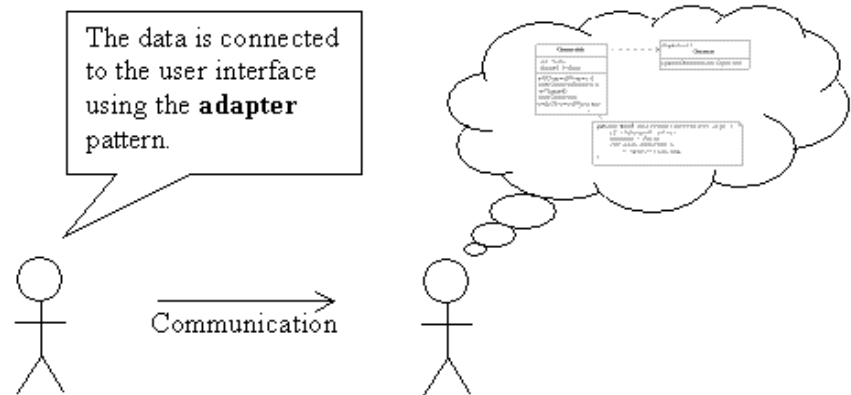
- *Aggiungere nuove tipologie di prodotti è difficile*

- devono essere create nuove realizzazioni per ogni factory.

How do design patterns help?

- I Design Pattern forniscono al progettista software
 - una soluzione codificata e consolidata per un problema ricorrente
 - un esempio di design modificabile (classi astratte e delega)
 - un modo per progettare software con caratteristiche predefinite
 - supporto alla progettazione di sistemi complessi
 - un vocabolario condiviso di supporto alla comunicazione delle caratteristiche del progetto all'interno di un team

*Quando un progettista dice
“questo è un adapter”
trasmette ai suoi interlocutori
in forma concisa e priva di
ambiguità una grossa
quantità di informazioni utili*



Riferimenti

- B. Bruegge et al, *Object-Oriented Software Engineering Using UML*, Patterns and Java, 2003.
- E. Gamma et.al., *Design Patterns*, 1994.
- <http://www.oose.globalse.org>
- M. Fowler, *Analysis Patterns: Reusable Object Models*, 1997

Nonfunctional Requirements may give a clue for the use of Design Patterns

- ♦ Read the problem statement again
- ♦ Use textual clues (similar to Abbot's technique in Analysis) to identify design patterns
- ♦ *Text*: “manufacturer independent”, “device independent”, “must support a family of products”
 - ♦ **Abstract Factory Pattern**
- ♦ *Text*: “must interface with an existing object”
 - ♦ **Adapter Pattern**
- ♦ *Text*: “must deal with the interface to several systems, some of them to be developed in the future”, “an early prototype must be demonstrated”
 - ♦ **Bridge Pattern**

Textual Clues in Nonfunctional Requirements

- ♦ *Text*: “complex structure”, “must have variable depth and width”
 - ♦ **Composite Pattern**
- ♦ *Text*: “must interface to a set of existing objects”
 - ♦ **Façade Pattern**
- ♦ *Text*: “must be location transparent”
 - ♦ **Proxy Pattern**
- ♦ *Text*: “must be extensible”, “must be scalable”
 - ♦ **Observer Pattern**
- ♦ *Text*: “must provide a policy independent from the mechanism”
 - ♦ **Strategy Pattern**

MVC

♦ *CONTESTO*

- ♦ *L'applicazione deve fornire una interfaccia grafica (GUI) costituita da più schermate, che mostrano vari dati all'utente. Inoltre le informazioni che devono essere visualizzate devono essere sempre quelle aggiornate*

♦ *PROBLEMA*

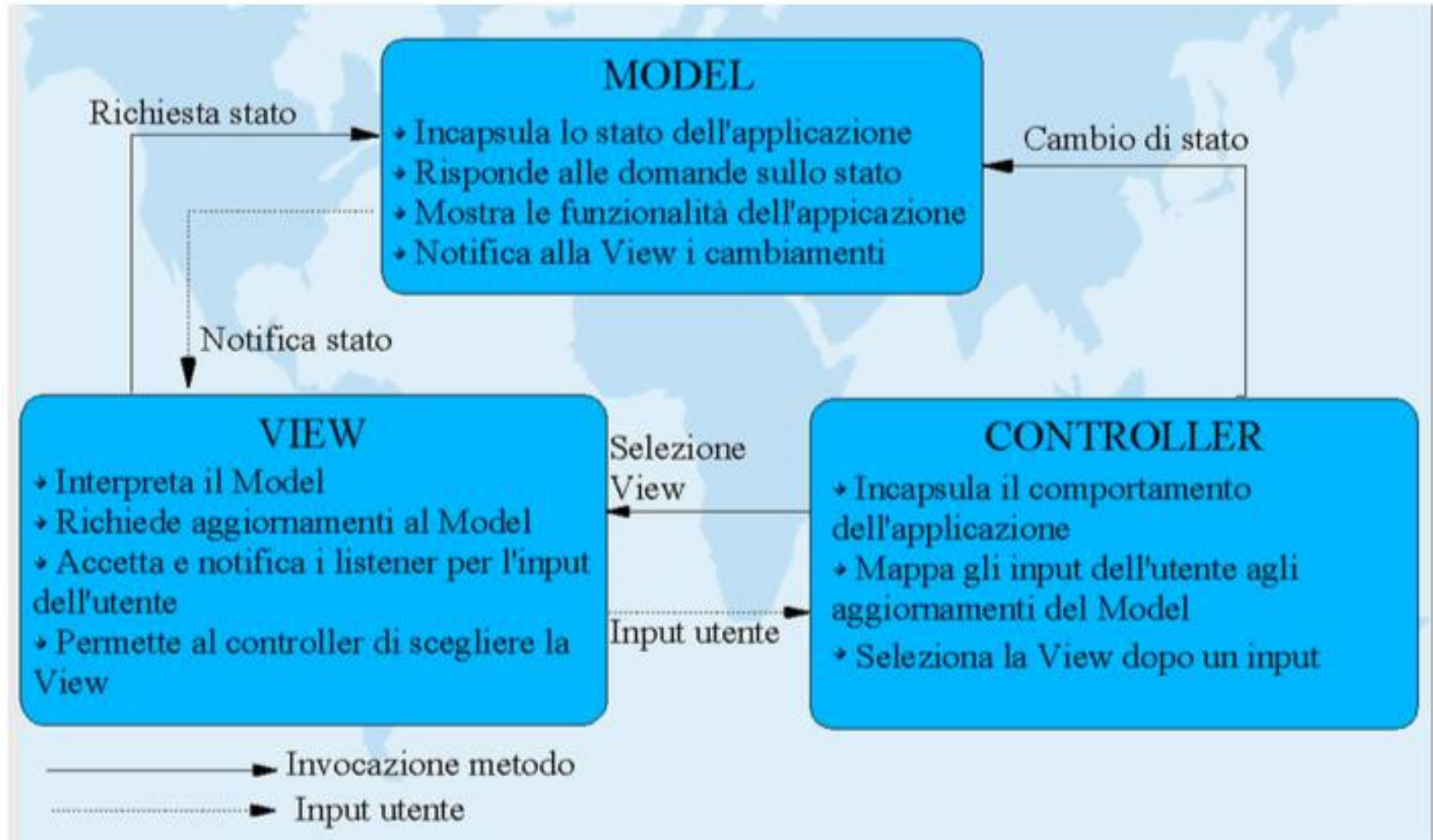
- ♦ *L'applicazione deve avere una natura modulare e basata sulle responsabilità, al fine di ottenere una vera e propria applicazione component - based.*

MVC

♦ *SOLUZIONE E STRUTTURA*

- ♦ *L'applicazione deve separare i componenti software che implementano il modello delle funzionalità di business, dai componenti che implementano la logica di presentazione e di controllo che utilizzano tali funzionalità. Vengono quindi definiti tre tipologie di componenti che soddisfano tali requisiti:*
 - *il Model, che implementa le funzionalità di business*
 - *la View: che implementa la logica di presentazione*
 - *il Controller: che implementa la logica di controllo*

MVC



Partecipanti e responsabilità

♦ **MODEL:**

il core dell'applicazione viene implementato dal Model, che incapsulando lo stato dell'applicazione definisce i dati e le operazioni che possono essere eseguite su questi.

- ♦ *Quindi definisce le regole di business per l'interazione con i dati, esponendo alla View ed al Controller rispettivamente le funzionalità per l'accesso e l'aggiornamento.*
- ♦ *Per lo sviluppo del Model quindi è vivamente consigliato utilizzare le tipiche tecniche di progettazione object oriented al fine di ottenere un componente software che astragga al meglio concetti importati dal mondo reale*
- ♦ *Il Model può inoltre avere la responsabilità di notificare ai componenti della View eventuali aggiornamenti verificatisi in seguito a richieste del Controller, al fine di permettere alle View di presentare agli occhi degli utenti dati sempre aggiornati.*

Partecipanti e responsabilità

♦ *VIEW:*

La logica di presentazione dei dati viene gestita solo e solamente dalla View.

- ♦ *deve gestire la costruzione dell' interfaccia grafica (GUI) - il mezzo mediante il quale gli utenti interagiranno con il sistema.*
- ♦ *Ogni GUI può essere costituita da schermate diverse che presentano più modi di interagire con i dati dell'applicazione.*
- ♦ *Per far sì che i dati presentati siano sempre aggiornati è possibile adottare due strategie note come "push model" e "pull model".*
- ♦

Partecipanti e responsabilità

- ♦ *Il push model adotta il pattern Observer, registrando le View come osservatori del Model.*
 - ♦ *Le View possono quindi richiedere gli aggiornamenti al Model in tempo reale grazie alla notifica di quest'ultimo.*
- ♦ *Benché push model rappresenti la strategia ideale, non è sempre applicabile.*
 - ♦ *Per esempio nell'architettura J2EE se le View che vengono implementate con JSP, restituiscono GUI costituite solo da contenuti statici (HTML) e quindi non in grado di eseguire operazioni sul Model.*
 - ♦ *In tali casi è possibile utilizzare il "pull Model" dove la View richiede gli aggiornamenti quando "lo ritiene opportuno".*
 - ♦ *Inoltre la View delega al Controller l'esecuzione dei processi richiesti dall'utente dopo averne catturato gli input e la scelta delle eventuali schermate da presentare.*

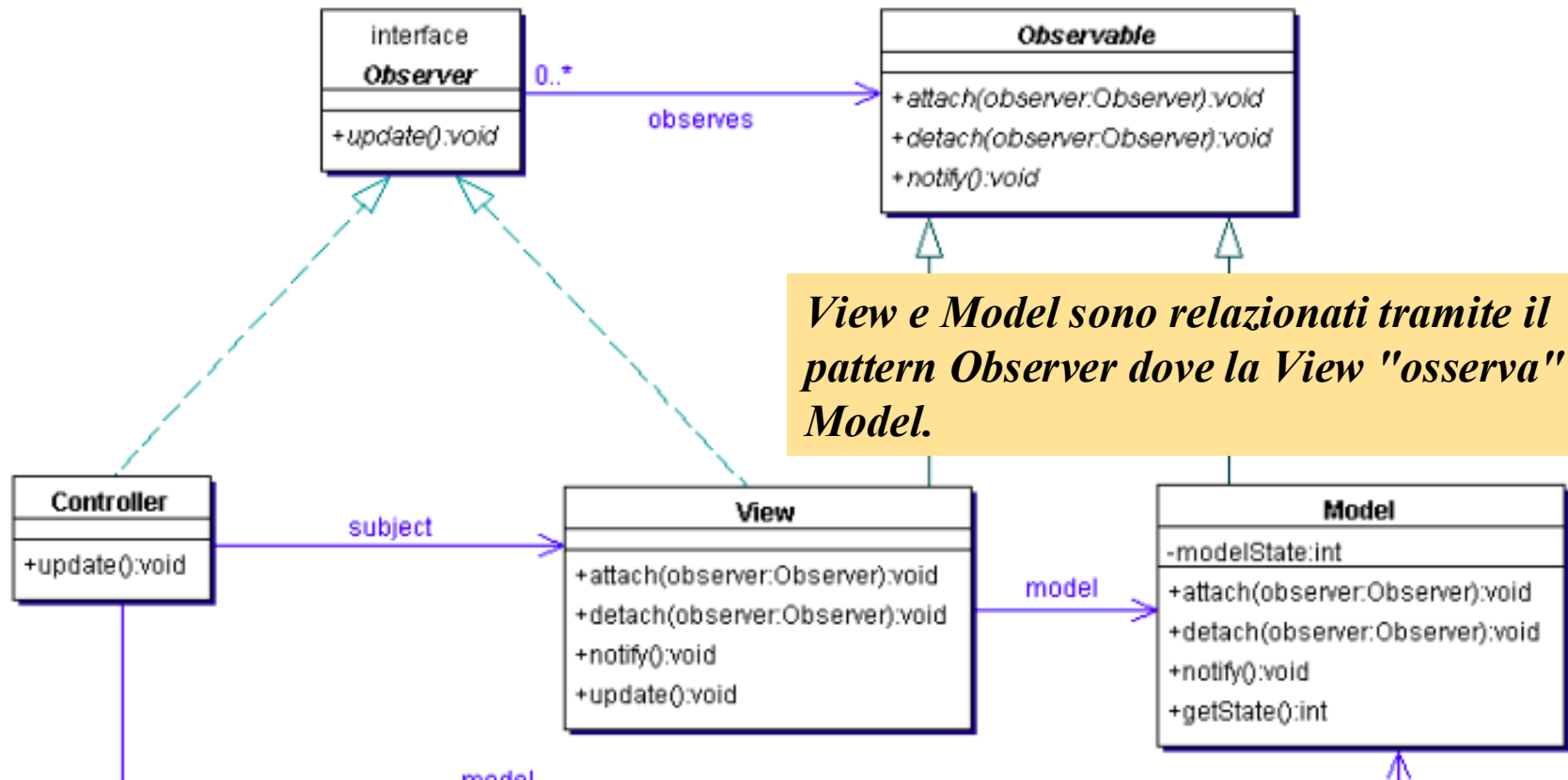
Partecipanti e responsabilità

♦ **CONTROLLER**

- ♦ *Questo componente ha la responsabilità di trasformare le interazioni dell'utente della View in azioni eseguite dal Model.*
- ♦ *Ma il Controller non rappresenta un semplice "ponte" tra View e Model.*
 - ♦ *Realizzando la mappatura tra input dell'utente e processi eseguiti dal Model e selezionando la schermate della View richieste, il Controller implementa la logica di controllo dell'applicazione.*

Diagramma delle classi

Mostra la “vera” natura del pattern MVC



View e Model sono relazionati tramite il pattern Observer dove la View "osserva" il Model.

Ma il pattern Observer caratterizza anche il legame tra View e Controller, ma in questo caso è la View ad essere "osservata" dal Controller. Ciò supporta la registrazione dinamica al runtime dei componenti.

Nota

- ♦ *Inoltre il pattern Strategy potrebbe semplificare la possibilità di cambiare la mappatura tra gli algoritmi che regolano i processi del Controller e le interazioni dell'utente con la View*
- ♦ *Che si tratti di un pattern che ne contenga altri, risulta abbastanza evidente. Ma nella sua natura originaria l'MVC comprendeva anche l'implementazione di altri pattern come il Factory Method per specificare la classe Controller di default per una View, il Composite per costruire View, ed il Decorator per aggiungere altre proprietà (per es. lo scrolling).*