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...

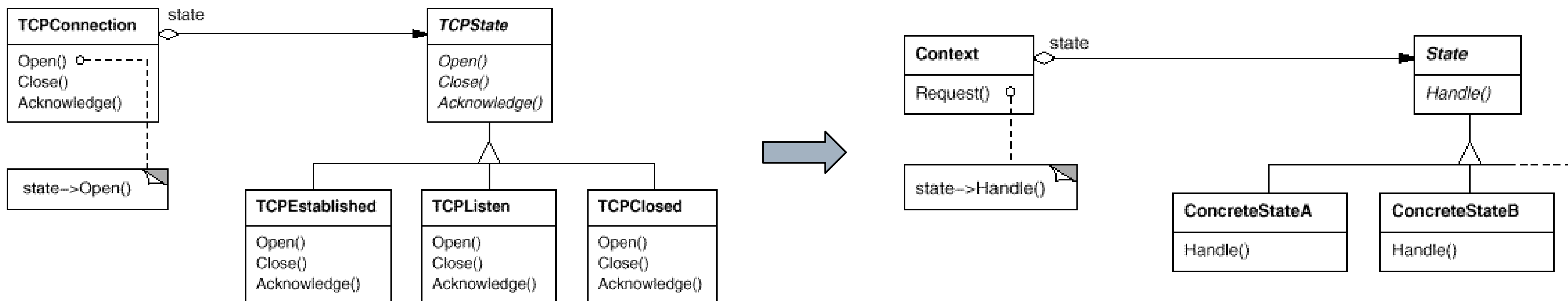
… "problemi"

- **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**):

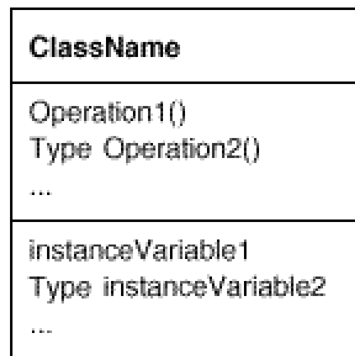
- **Class pattern**
  - si focalizzano sulle relazioni tra classi e sottoclassi
  - tipicamente si riferiscono a situazioni statiche (per es., relazioni espresse attraverso l'ereditarietà)

- **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

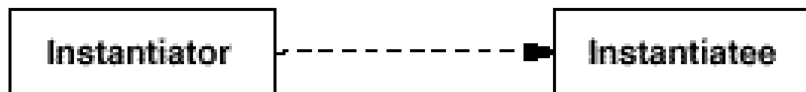
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State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

Interpretazione dei Pattern

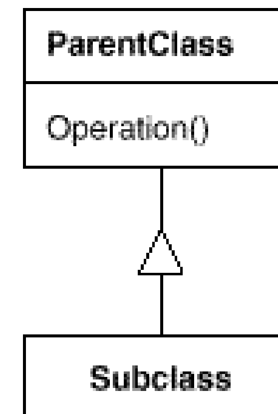


Classe

Aggregazione

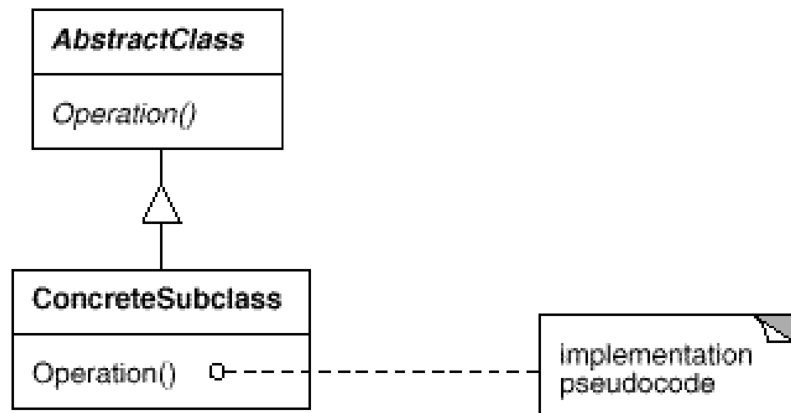


Indica una classe che istanzia oggetti di un'altra classe

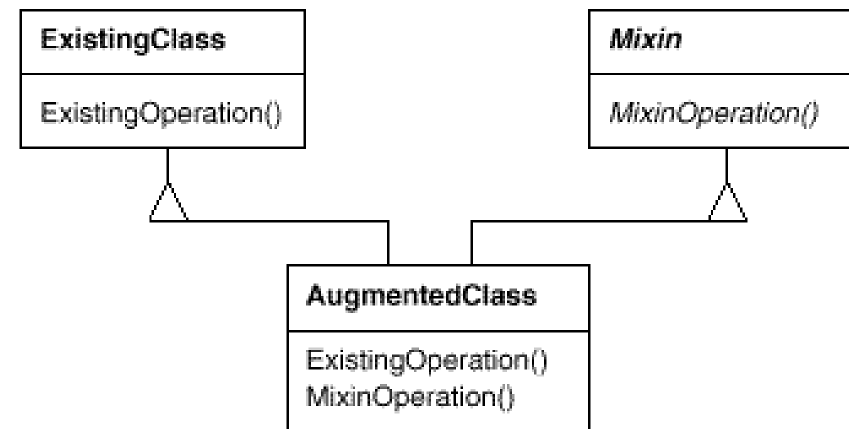


Ereditarietà

Interpretazione dei Pattern (ii)



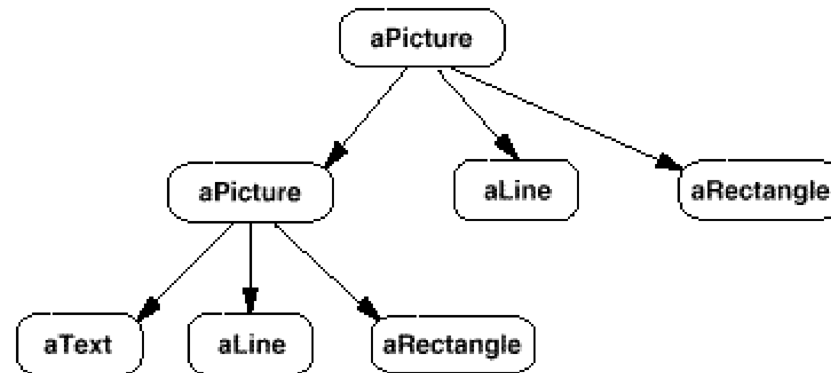
Classe astratta



Ereditarietà multipla

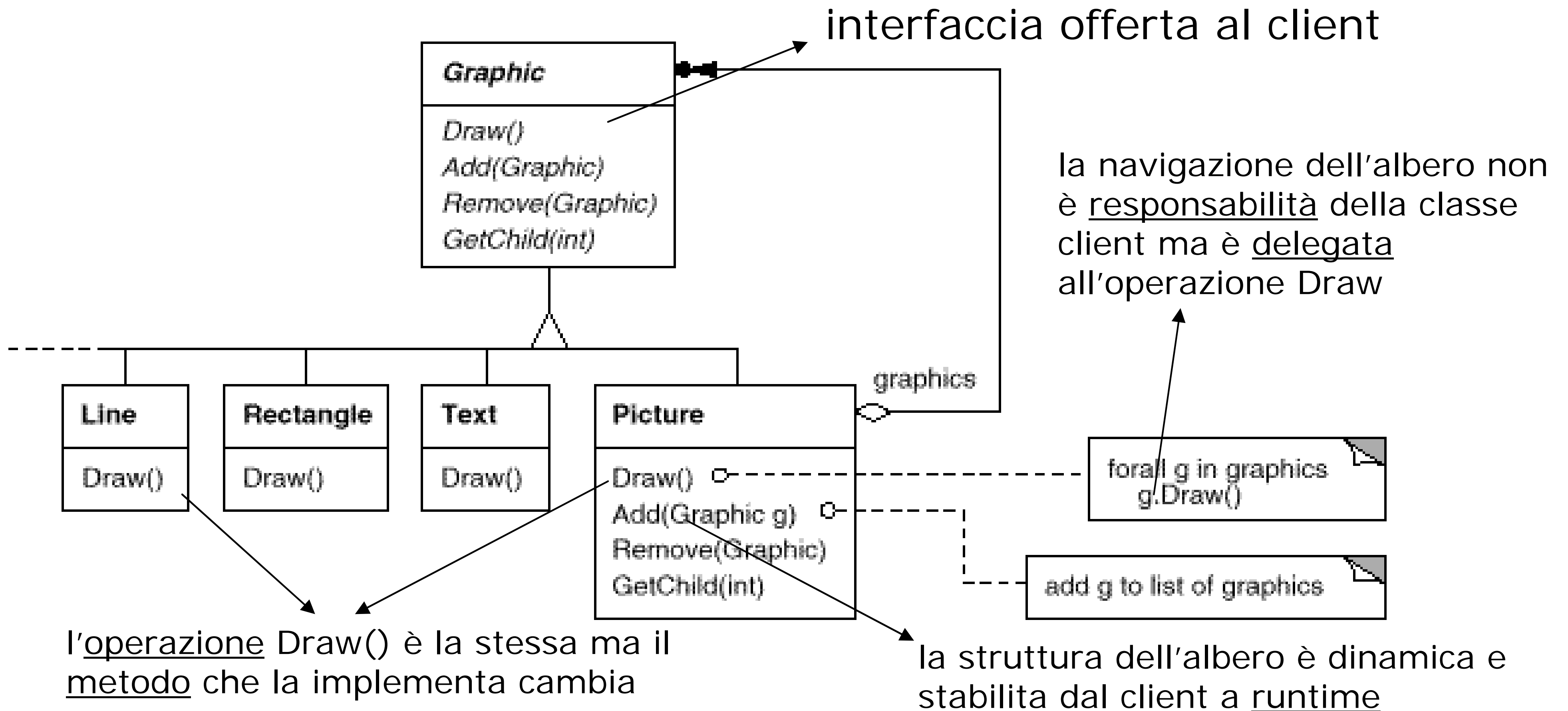
Esempio : Composite

- 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 <u>client</u> 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

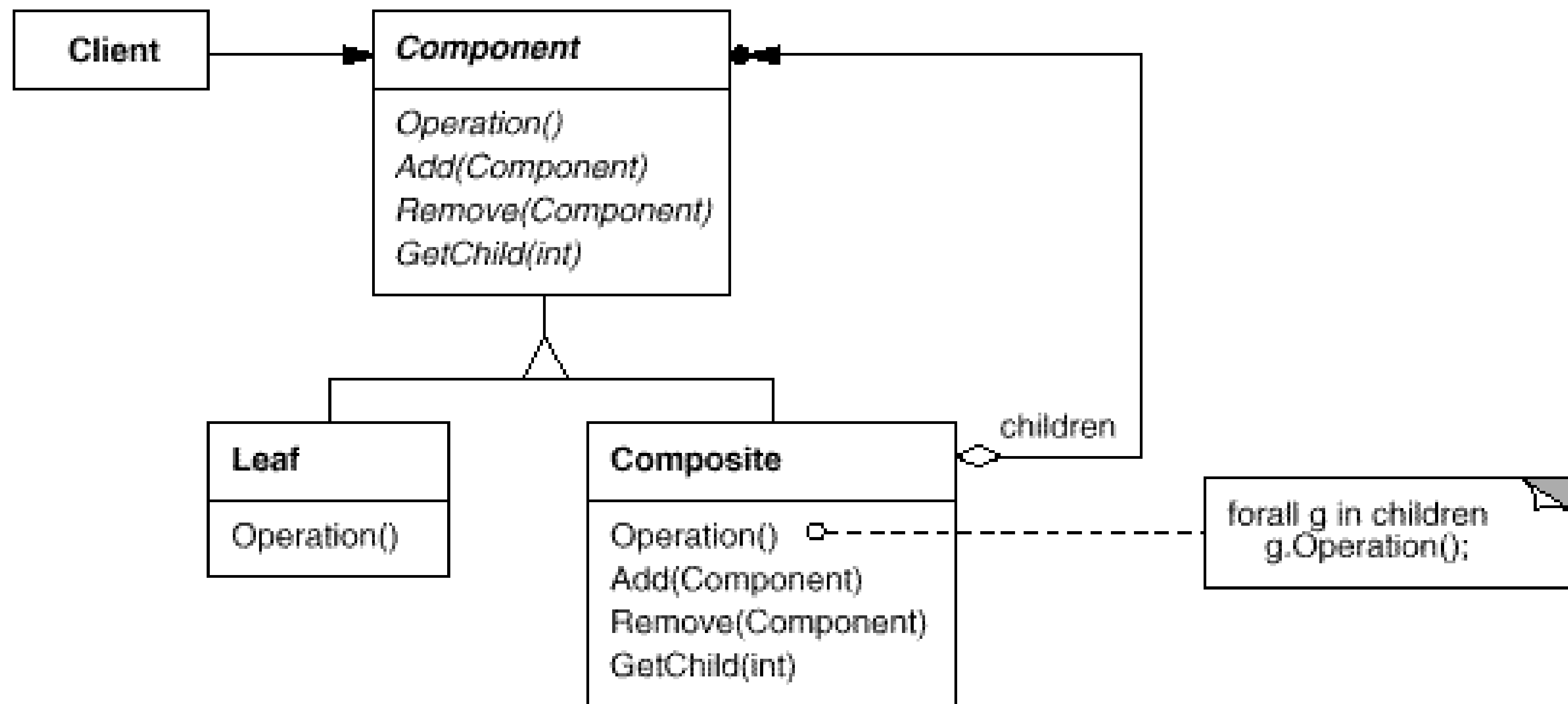


quali sono i meccanismi orientati agli oggetti offerti da un linguaggio come il C++ per implementare il pattern Composite?

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

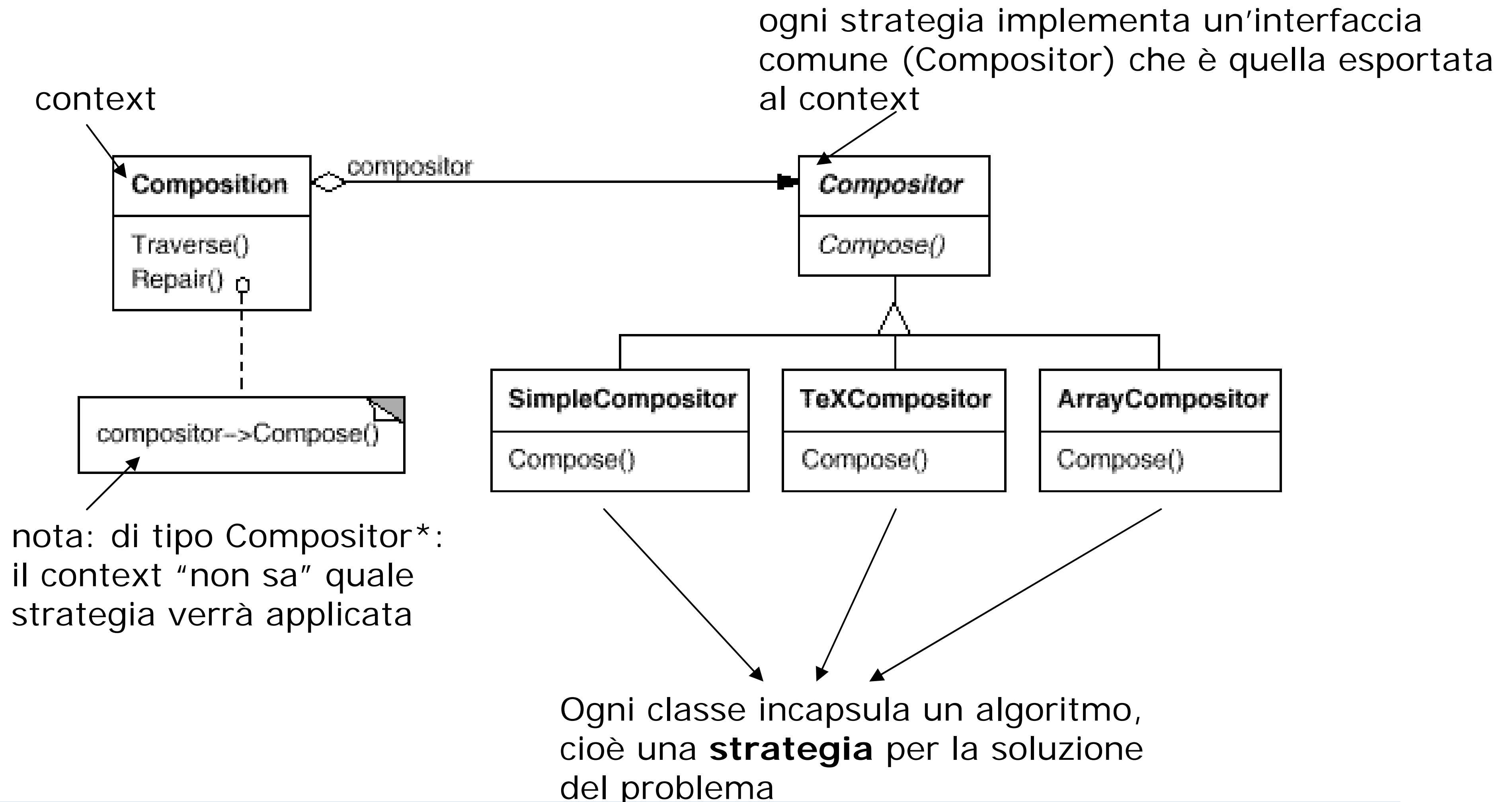
Strategy – Esempio

Algoritmi per suddividere il testo in righe in un editor di testo

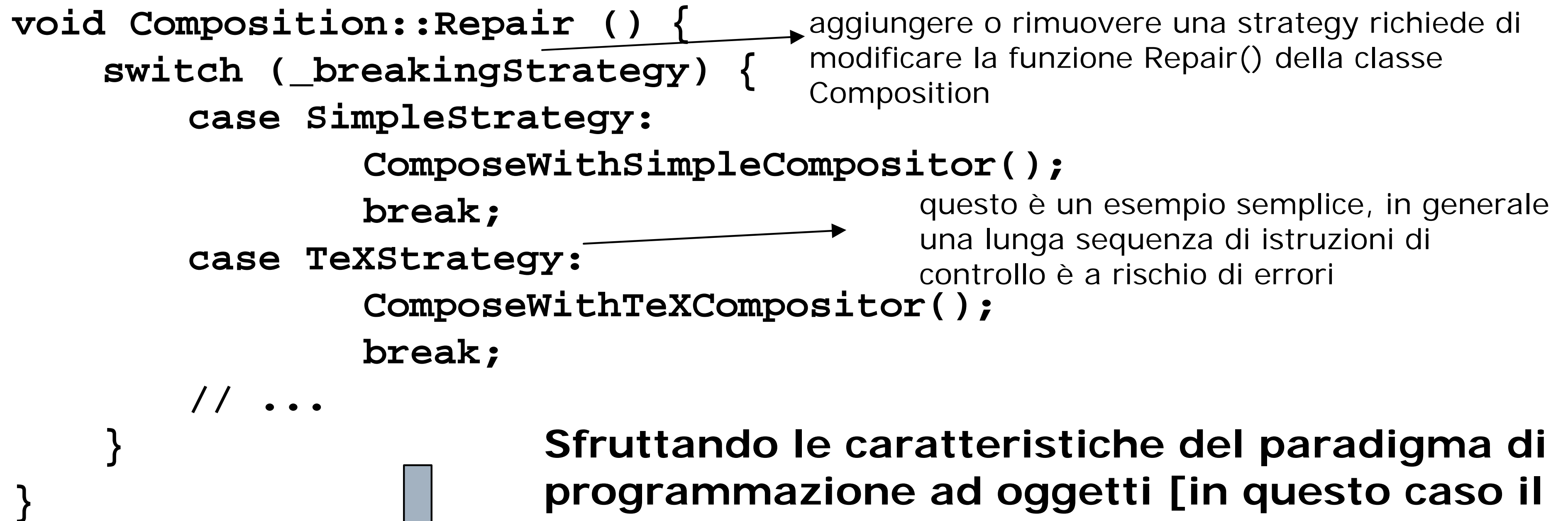
- Ne esistono di varia natura, più o meno complessi (es. numero fisso di parole, lunghezza fissa di una riga, etc)
- Includere il codice degli algoritmi nel client* non è desiderabile
  - Client più complessi e più difficili da manutenere
  - Diversi algoritmi sono adatti per diverse occasioni (ridondanza)
  - Difficoltà di inserire nuovi algoritmi quando questi sono parte del client

*oggetto di alto livello che gestisce elementi grafici

Strategy – Soluzione



Strategy – Soluzione (ii)



Sfruttando le caratteristiche del paradigma di programmazione ad oggetti [in questo caso il polimorfismo] si può ottenere una notevole semplificazione del codice

```
void Composition::Repair () {
    _compositor->Compose();
}
```

in generale è necessario specificare come avviene la comunicazione tra Compositor e context

di tipo Strategy, interfaccia condivisa da tutte le classi che la implementano

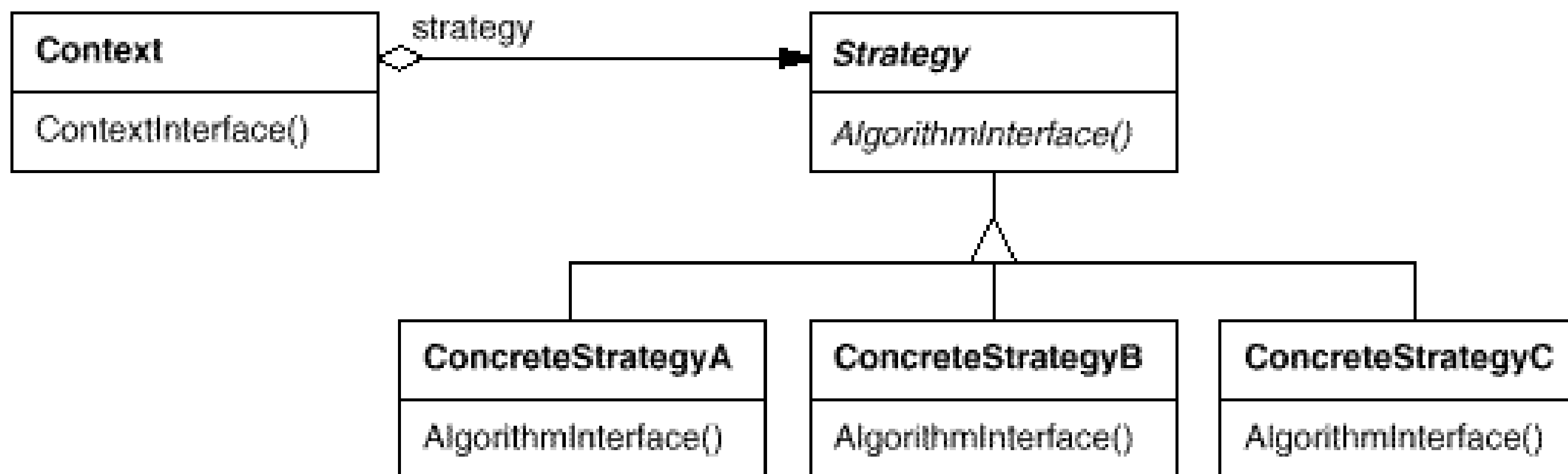
Strategy – Soluzione (iii)

```
Composition* quick = new Composition(new
    SimpleCompositor);
Composition* slick = new Composition(new TeXCompositor);
Composition* iconic = new Composition(new
    ArrayCompositor(100));
```

- ad ogni oggetto di tipo compositor viene associata una strategia di formattazione del testo (ad es. contestualmente alla creazione)
- il resto del codice rimane trasparente rispetto a questa scelta: cambiare strategia richiede di modificare una riga di codice

Strategy (object/behavioral)

- Definisce una famiglia di algoritmi, incapsulando ognuno di essi e rendendoli interscambiabili
- Permette che l'algoritmo cambi in maniera trasparente rispetto al client che lo usa



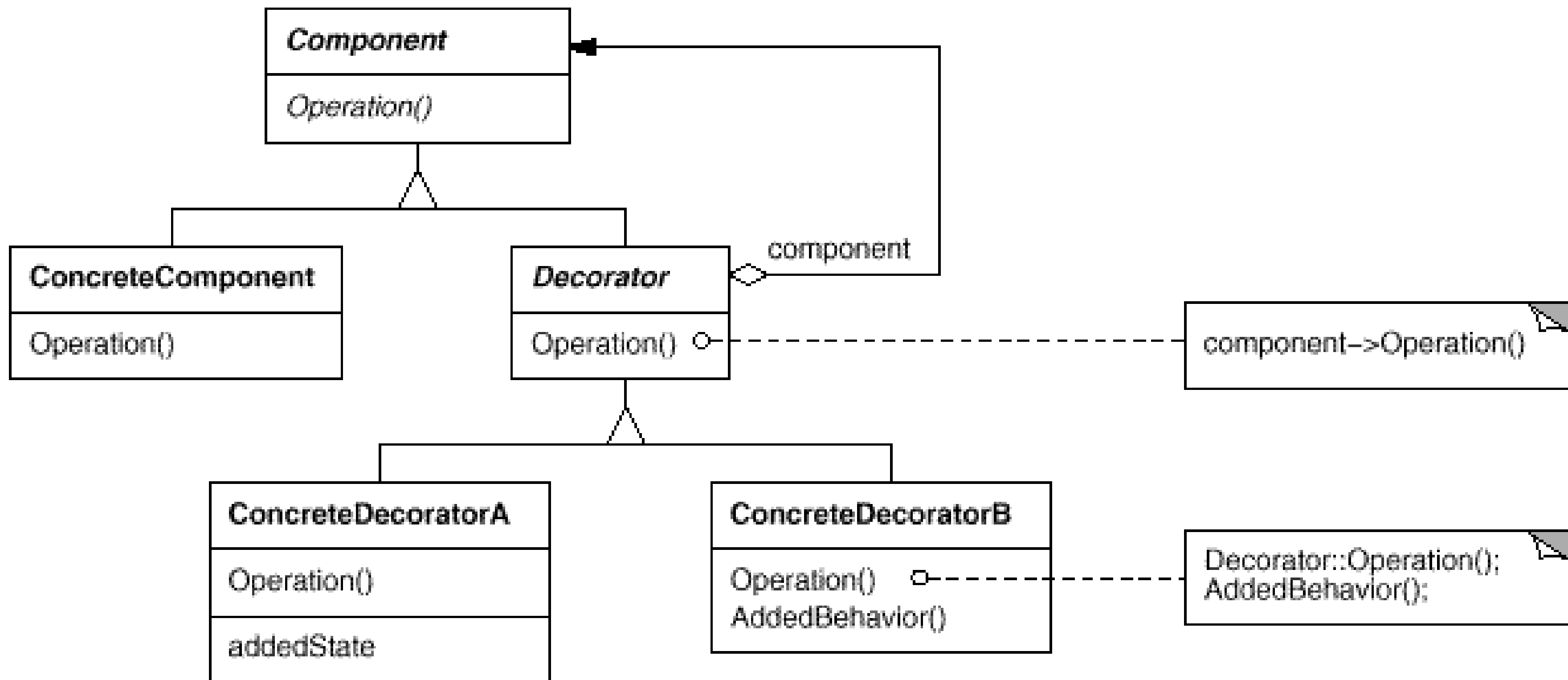
Decorator (object / structural)

- Permette di assegnare una o più responsabilità addizionali ad un <u>oggetto</u> in maniera dinamica
- Rappresenta una flessibile alternativa al subclassing per estendere le funzionalità di un oggetto

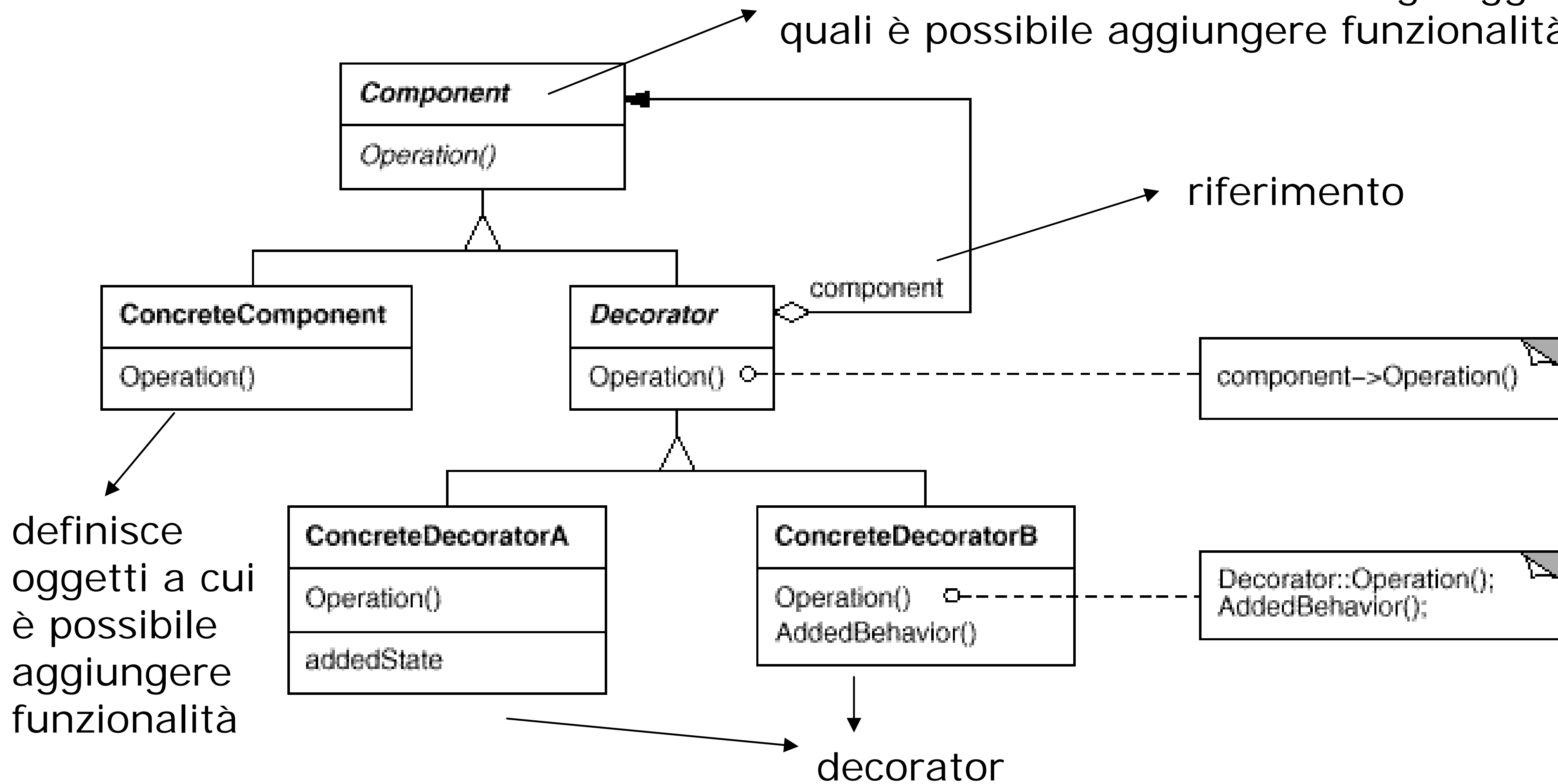
Come si possono estendere le funzionalità di un oggetto?

1. **Specializzazione**: faccio derivare dalla classe a cui appartiene l'oggetto un'ulteriore classe con le funzionalità aggiuntive
   - PROBLEMA: relazione statica -> un client non può controllare quando e come aggiungere tali funzionalità
   - PROBLEMA: la gerarchia di classi può diventare complessa se si prevedono varie funzionalità/combinazioni
2. **Classi wrapper (Decorator)**: inglobare l'oggetto all'interno di un altro oggetto (il decorator) che aggiunge le nuove funzionalità

Decorator (ii)



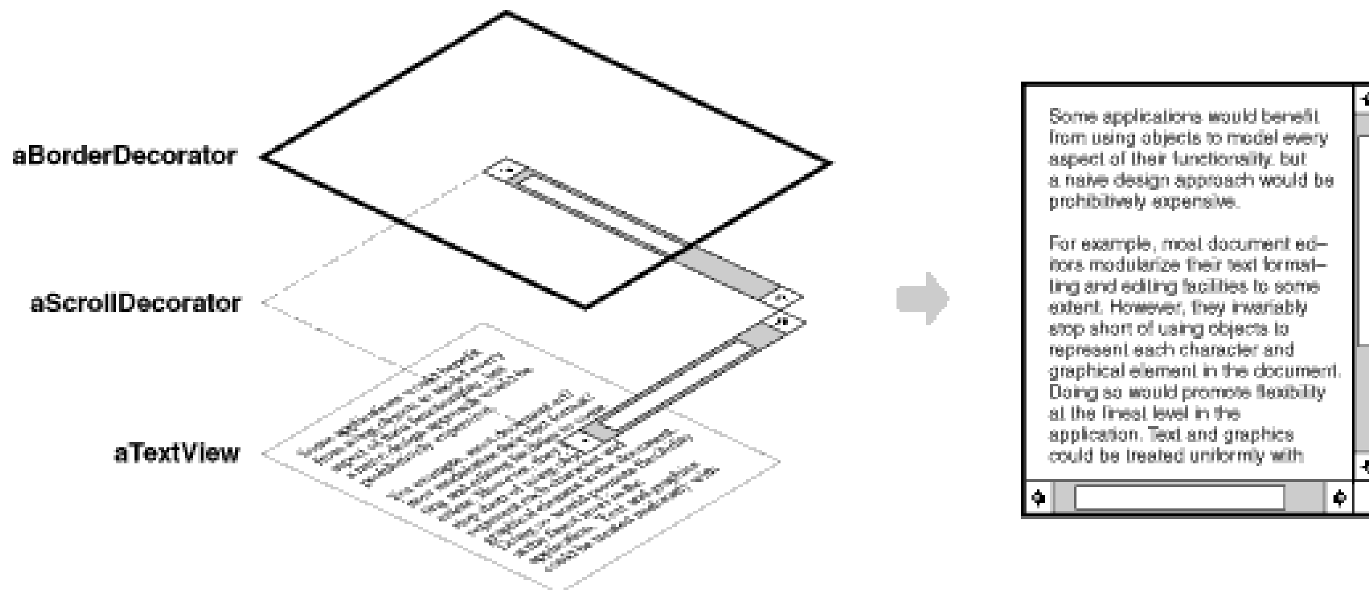
Decorator (iii)



Decorator – Esempio

GUI toolkit

Permette di assegnare ad un elemento grafico altri elementi aggiuntivi (opzionali) come scrollbar, bordi, ecc., in maniera ***dinamica***



Decorator – Esempio (ii)

- Oggetto di tipo TextView (eredita da VisualComponent) che visualizza il testo all'interno della finestra (operazione `draw()`)
  - Di default non ha né scrollbar, né bordi:

```
class VisualComponent {
   public:
          VisualComponent();
          virtual void Draw();
          // ...
};
```

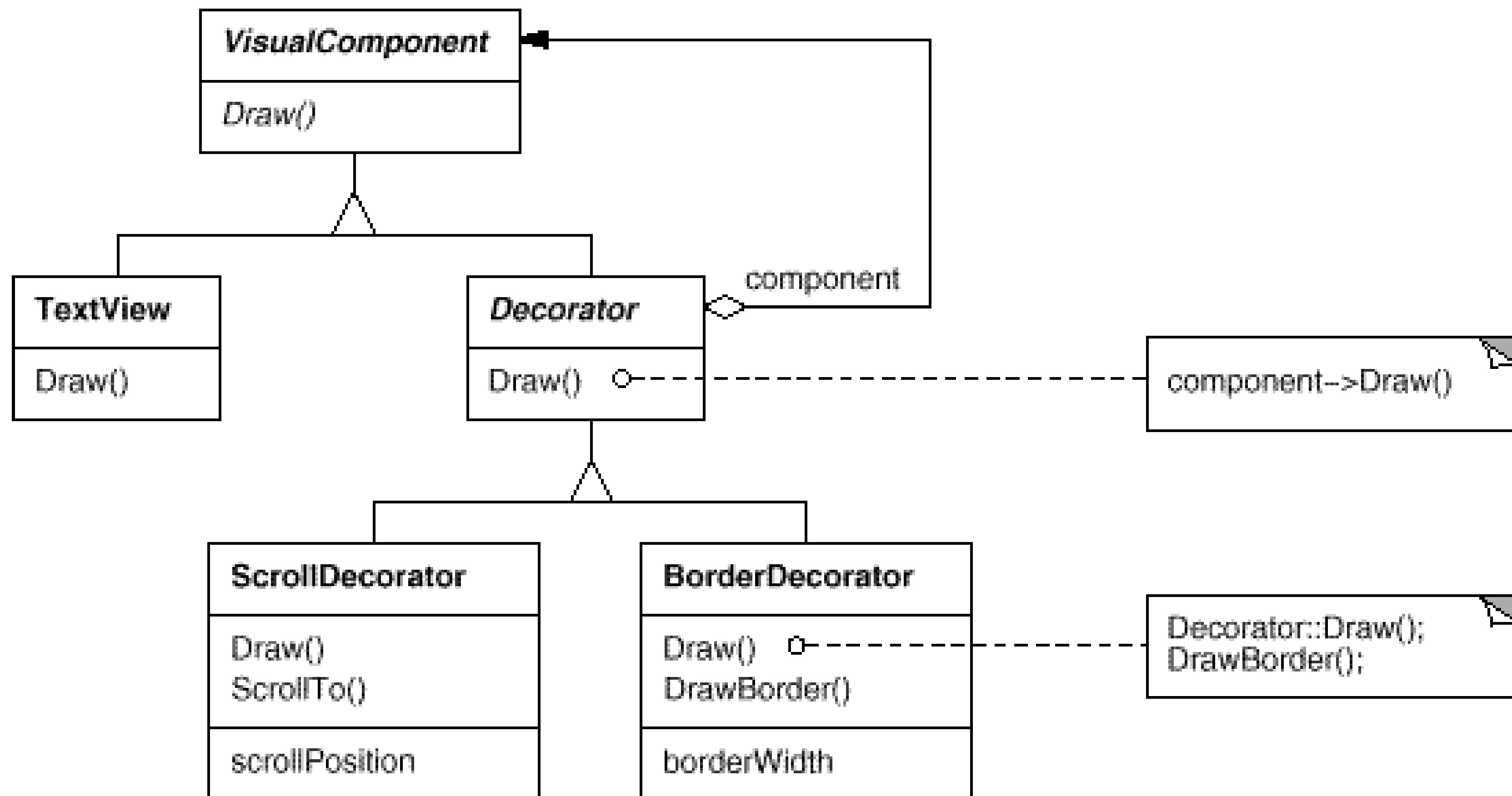
```
class TextView:public VisualComponent {
   public:
          TextView();
          virtual void Draw();
          // ...
};
```

Vogliamo aggiungere:

- Possibilità di aggiungere una scrollbar (operazione aggiuntiva `scrollTo()` e informazione di stato aggiuntiva `scrollPosition`)
- Possibilità di aggiungere un bordo (operazione aggiuntiva `drawBorder()` e informazione di stato aggiuntiva `borderWidth`)

Applicare il design pattern Decorator e scrivere il codice relativo in C++

Decorator – Soluzione



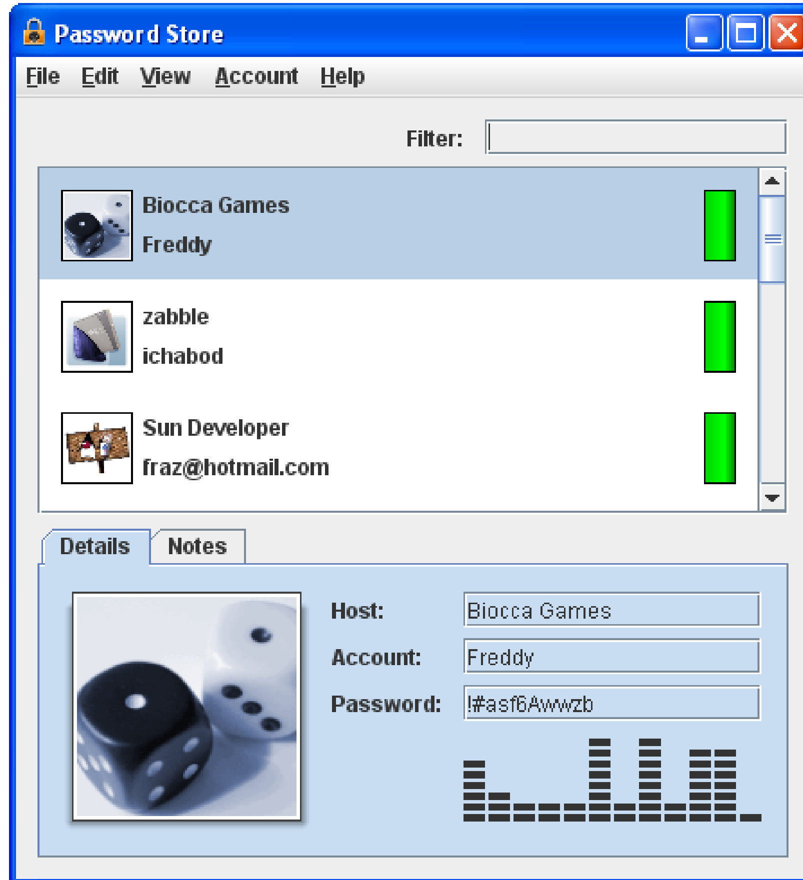
Decorator – Soluzione (ii)

```
class VisualComponent {
    public:
        VisualComponent();
        virtual void Draw();
        // ...
};

class Decorator : public VisualComponent {
    public:
        Decorator(VisualComponent*);
        virtual void Draw();
        virtual void Resize();
        // ...
    private:
        VisualComponent* _component;
};

void Decorator::Draw () {
    _component->Draw();
}
```

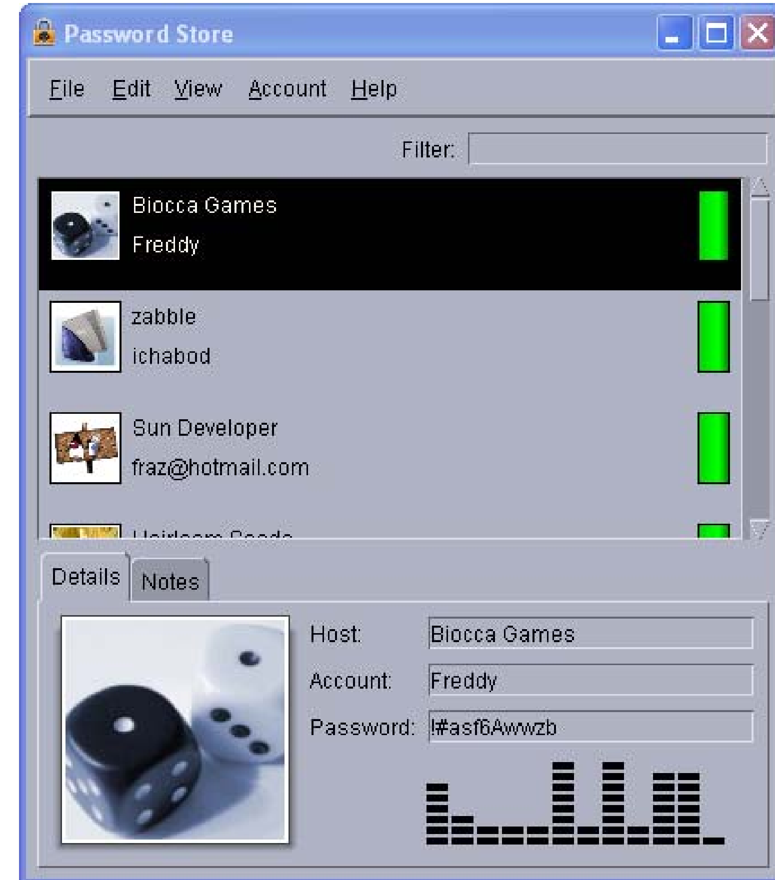
Decorator – Soluzione (iii)

```
class BorderDecorator : public Decorator {
    public:
        BorderDecorator(VisualComponent*, int borderWidth);
        virtual void Draw();
    private:
        void DrawBorder(int);
    private:
        int _width;
};

void BorderDecorator::Draw () {
        Decorator::Draw();
        DrawBorder(_width);
}
```

Abstract Factory – Esempio

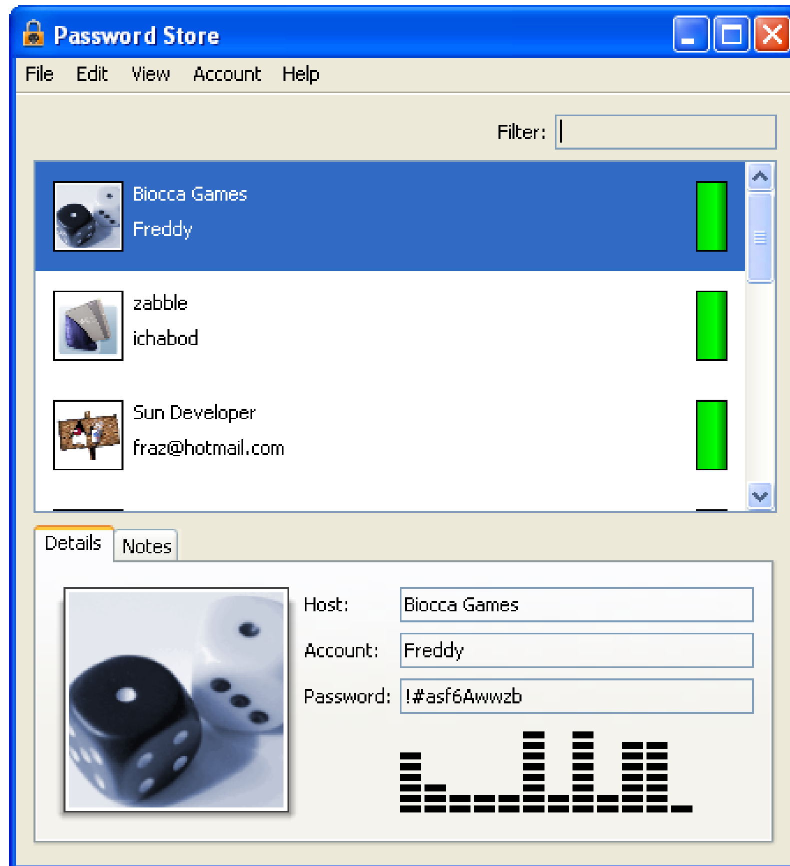


"Java" look and feel

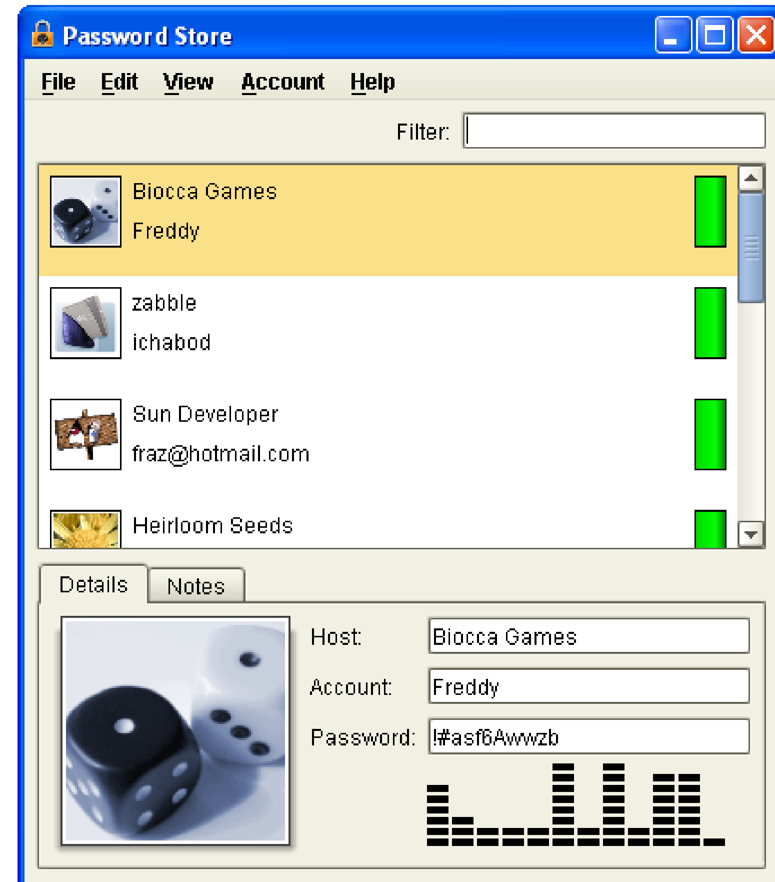


"Windows" look and feel

Abstract Factory – Esempio

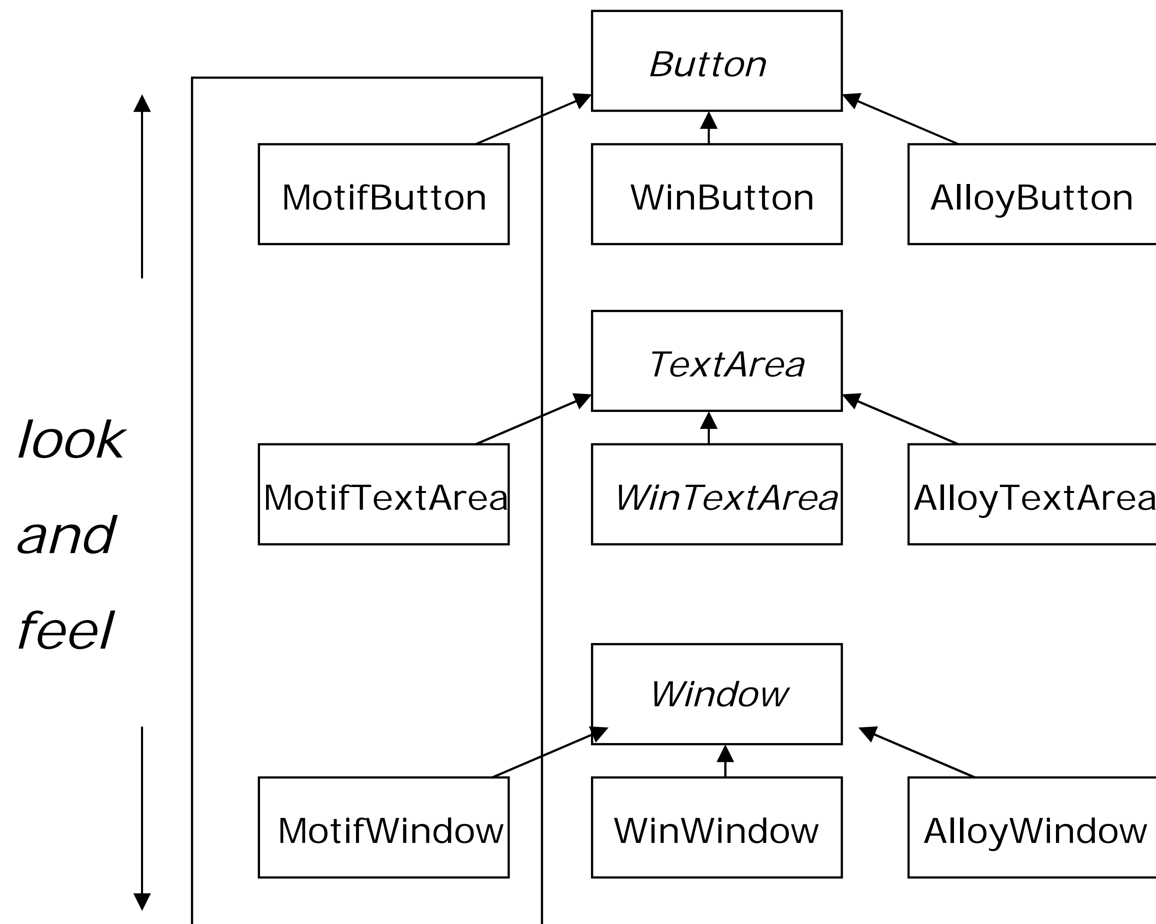


“Motif” look and feel



“Alloy” look and feel

Abstract Factory – Esempio



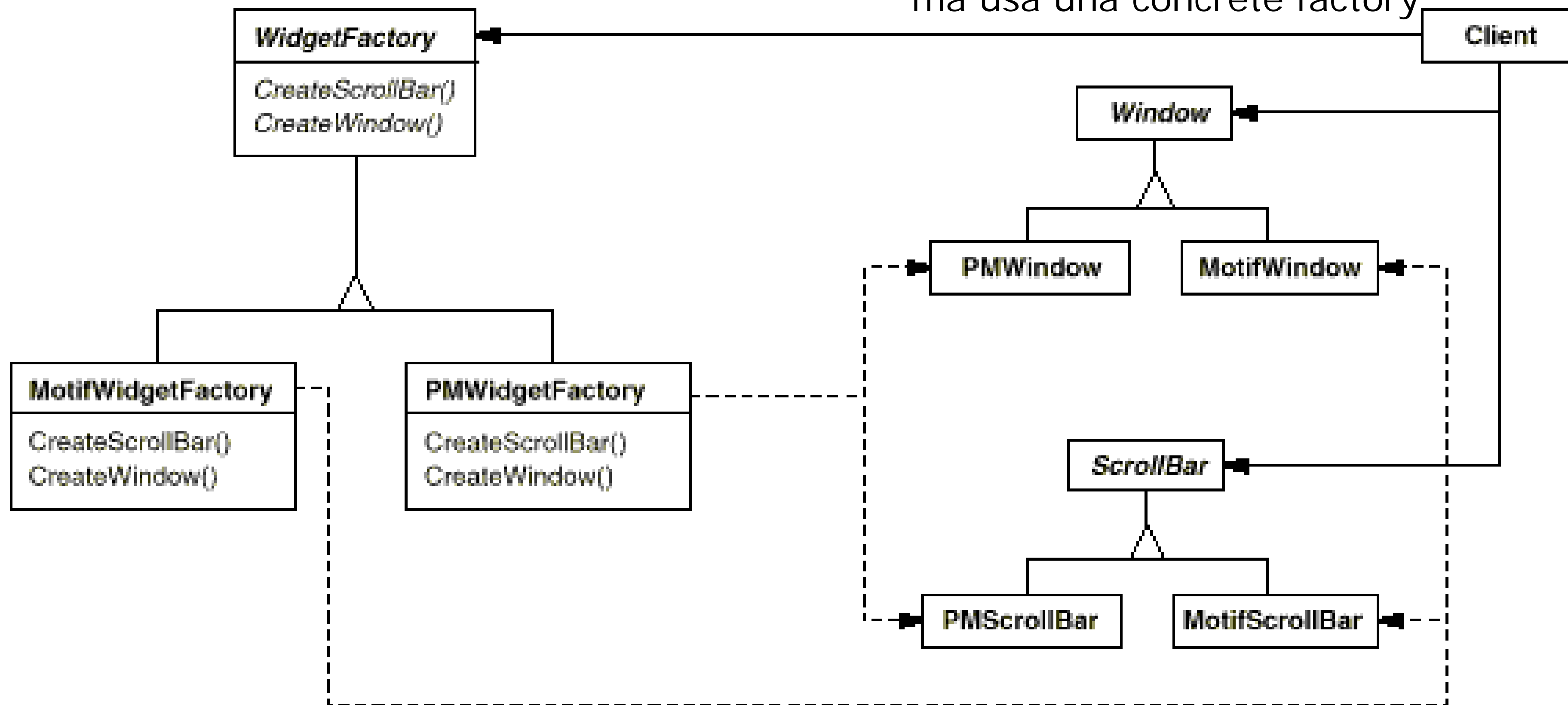
Abstract Factory – Esempio

- Una GUI toolkit supporta diversi **“look-and-feel*”**, come Motif e Presentation Manager (le FAMIGLIE DI PRODOTTI)
- Diversi look-and-feel definiscono un diverso aspetto e diversi comportamenti per “widgets” (i PRODOTTI) come:
  - Scrollbars
  - Windows
  - Buttons
  - TextFields .. etc
- Ci aspettiamo che impostare un certo look and feel per la GUI imponga un vincolo sull’uniformità dello stile dei vari componenti (es. se i Button sono “stile” Motif allora la scrollbar deve essere “Motif” …) vorremmo che l’aspetto dei vari componenti sia coerente e che controllare il look and feel dei singoli oggetti da creare non sia compito del programmatore)
- In generale vorremmo scegliere una volta per tutte la famiglia di prodotti da utilizzare e delegare il compito di instanziare volta per volta l’oggetto di tipo giusto

*ad es. forma dei componenti, tonalità, font etc

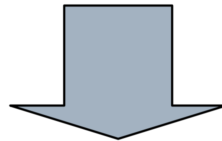
Abstract Factory – Esempio

non chiama i costruttori per i vari oggetti, ma usa una concrete factory



Abstract Factory – Esempio

```
class GUI {
    //...
        void Create_gui(){
//...
button=new MotifButton();
textArea=new MotifTextArea("text");
window=new MotifWindow();
//..
};
```

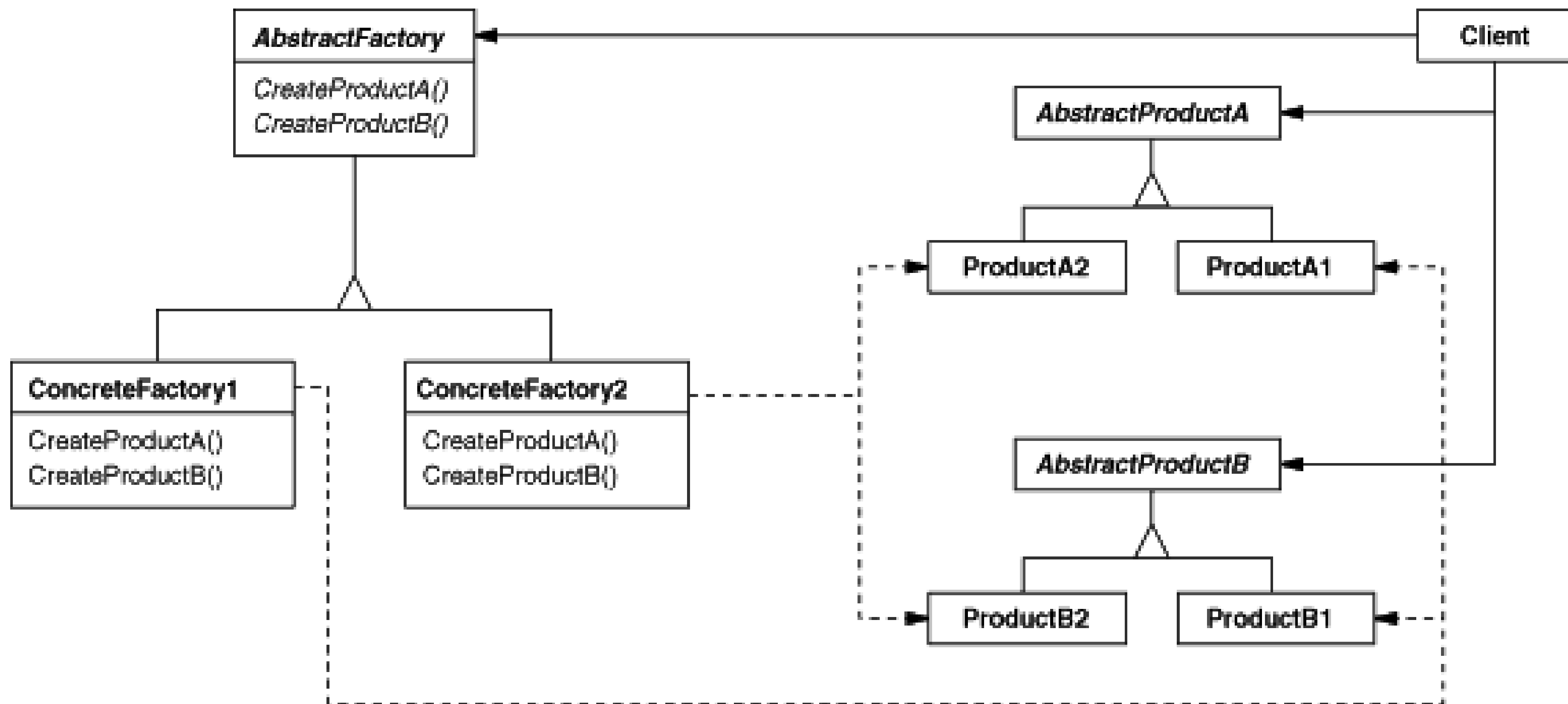


```
class GUI {
    //...
    WidgetFactory* factory;
    void Create_gui(){
//...
button=factory->createButton();
textArea=factory->createTextArea("text");
window=factory->createWindow();
//..
};
```

Abstract Factory (object / creational)

- Fornisce un’interfaccia per creare famiglie di oggetti correlati/dipendenti senza specificare le loro classi concrete
- Quando usare Abstract Factory:
  - Un sistema deve essere indipendente da come i suoi prodotti sono creati, composti e rappresentati
  - Un sistema deve essere configurato per **una** di diverse famiglie di prodotti disponibili
  - Una famiglia di oggetti relativi a prodotti correlati è progettata per essere usata nella sua totalità
  - Si vuole fornire una libreria di classi di prodotti, e si vogliono rivelare solo le interfacce, non le loro implementazioni

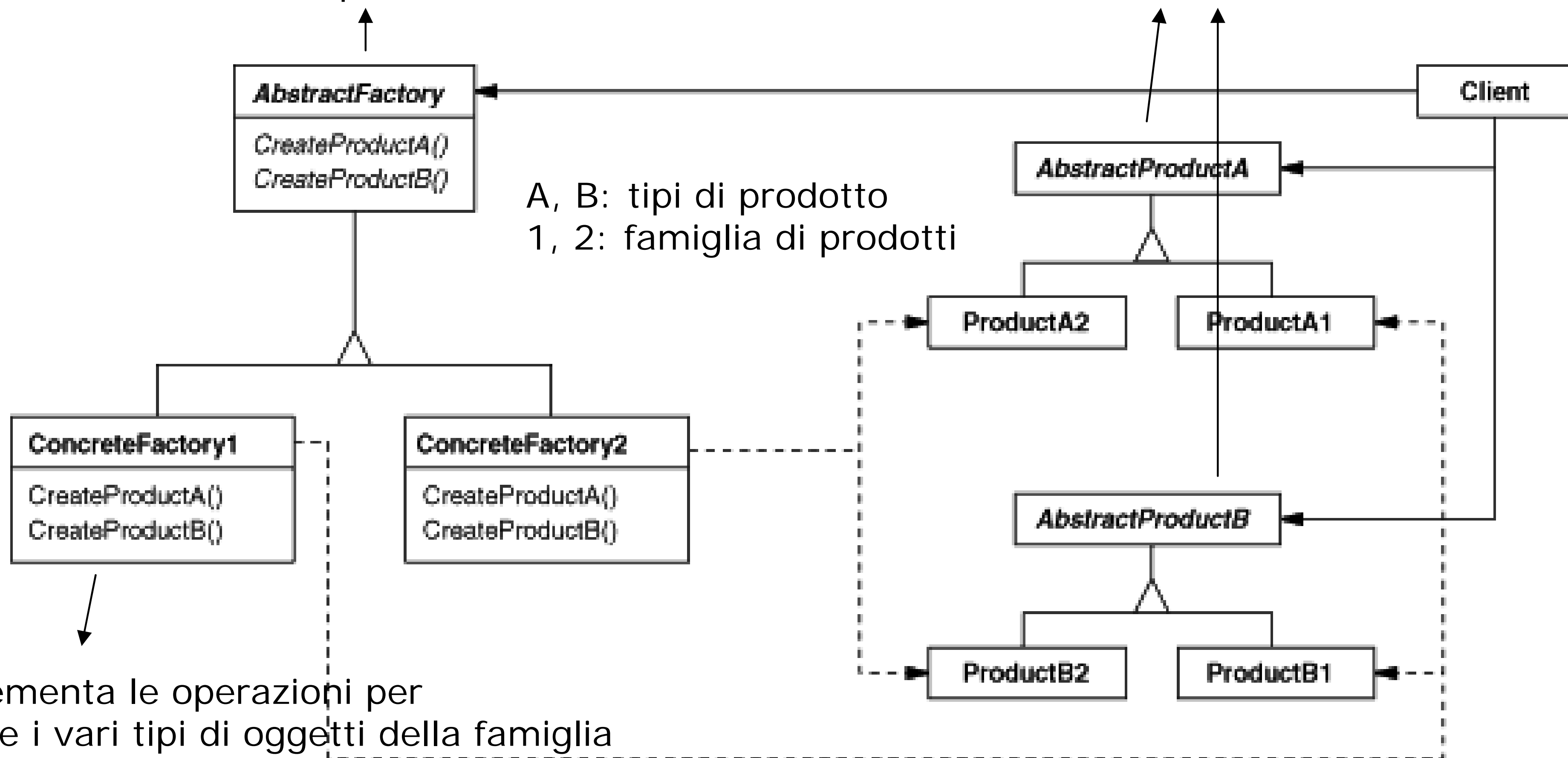
Abstract Factory (ii)



Abstract Factory (iii)

Dichiara un interfaccia per operazioni
di creazione di prodotti

Interfaccia per una tipologia di prodotti



Abstract Factory: Pros

- Isola le classi concrete
  - I client non devono sapere niente delle classi concrete che useranno, neanche al momento dell'istanziazione degli oggetti
- E' facile cambiare famiglia di prodotto
  - Basta cambiare 1 linea di codice che riguarda la creazione della factory
- Promuove la consistenza tra i prodotti
  - I prodotti sono organizzati in famiglie. I prodotti di una famiglia sono coordinati per lavorare insieme

Abstract Factory: Cons

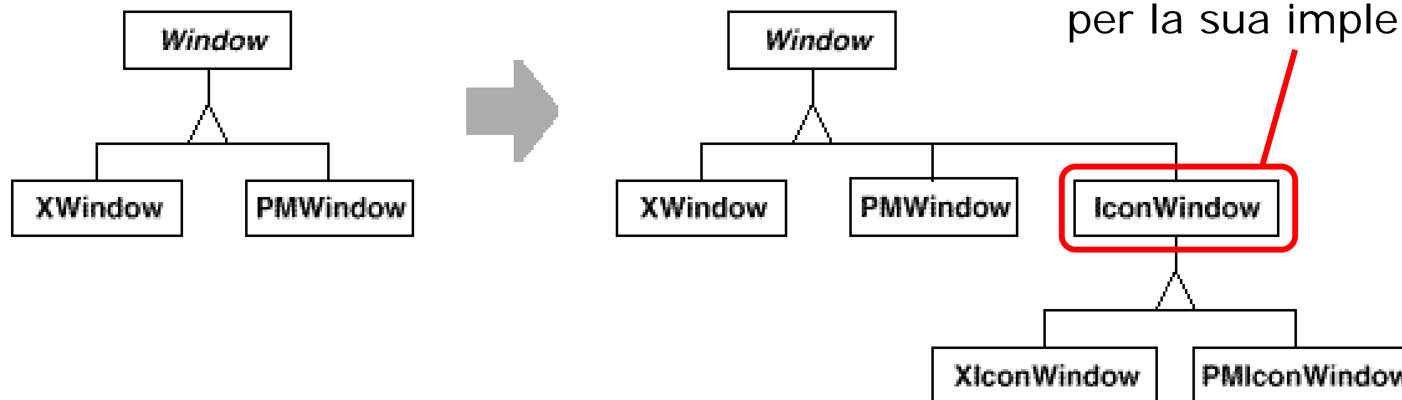
- Supportare l'inserimento di un nuovo prodotto in una famiglia è difficile
  - Può richiedere cambiamenti all'interfaccia dell'Abstract Factory e alle sue sottoclassi
  - Richiede l'aggiunta di una nuova classe ConcreteFactory e di nuovi AbstractProducts e Products
- La creazione di oggetti non avviene nel modo standard
  - I client devono sapere che devono usare la factory invece del costruttore per istanziare nuovi oggetti
  - Altrimenti la correttezza dell'implementazione non è garantita

Bridge

- Problema: un'astrazione può avere una implementazione tra tante diverse fra loro
  - Tipica soluzione: **ereditarietà**
    Una classe astratta definisce l'interfaccia dell'astrazione e le sottoclassi realizzano ciascuna una singola implementazione

  Tale soluzione ha però delle limitazioni:

Per ogni tipologia di astrazione aggiuntiva devo avere 2 classi per la sua implementazione



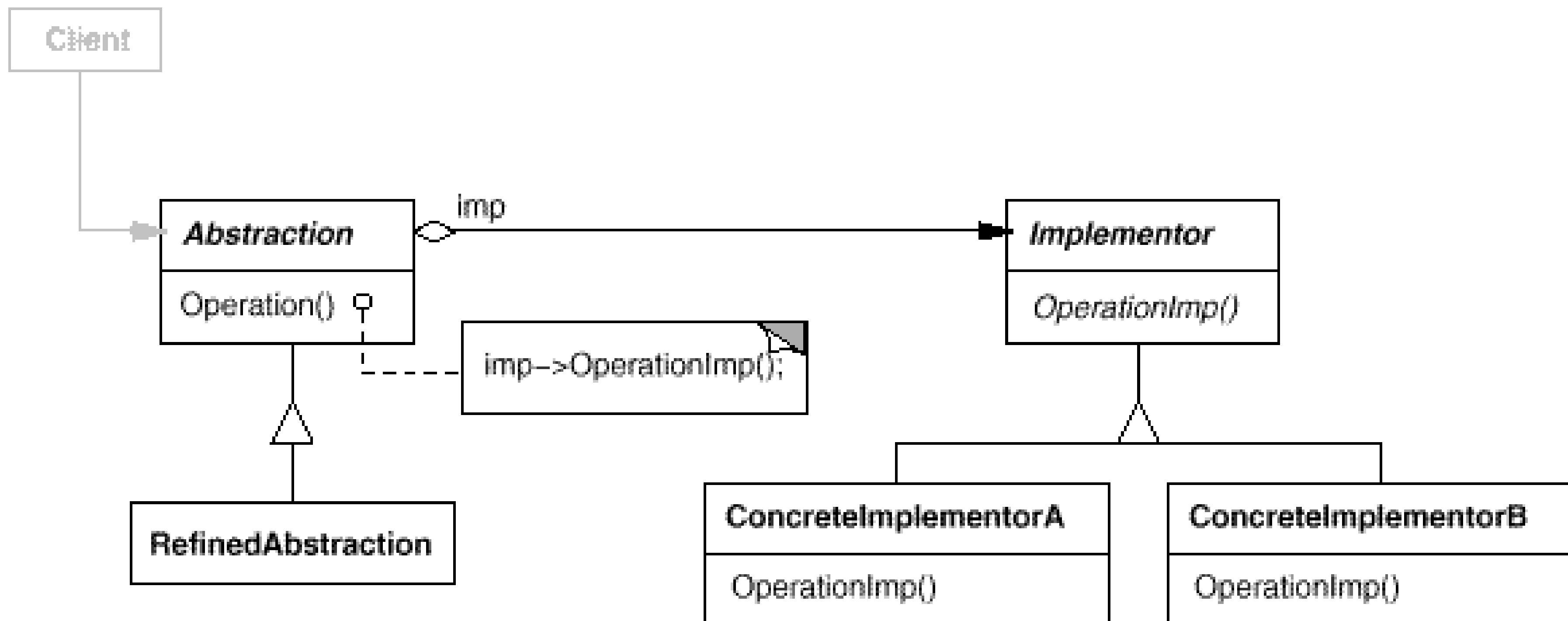
Nota: implicitamente astrazione e implementazione sono mescolate nella stessa gerarchia ...

Bridge (ii)

Limitazioni:

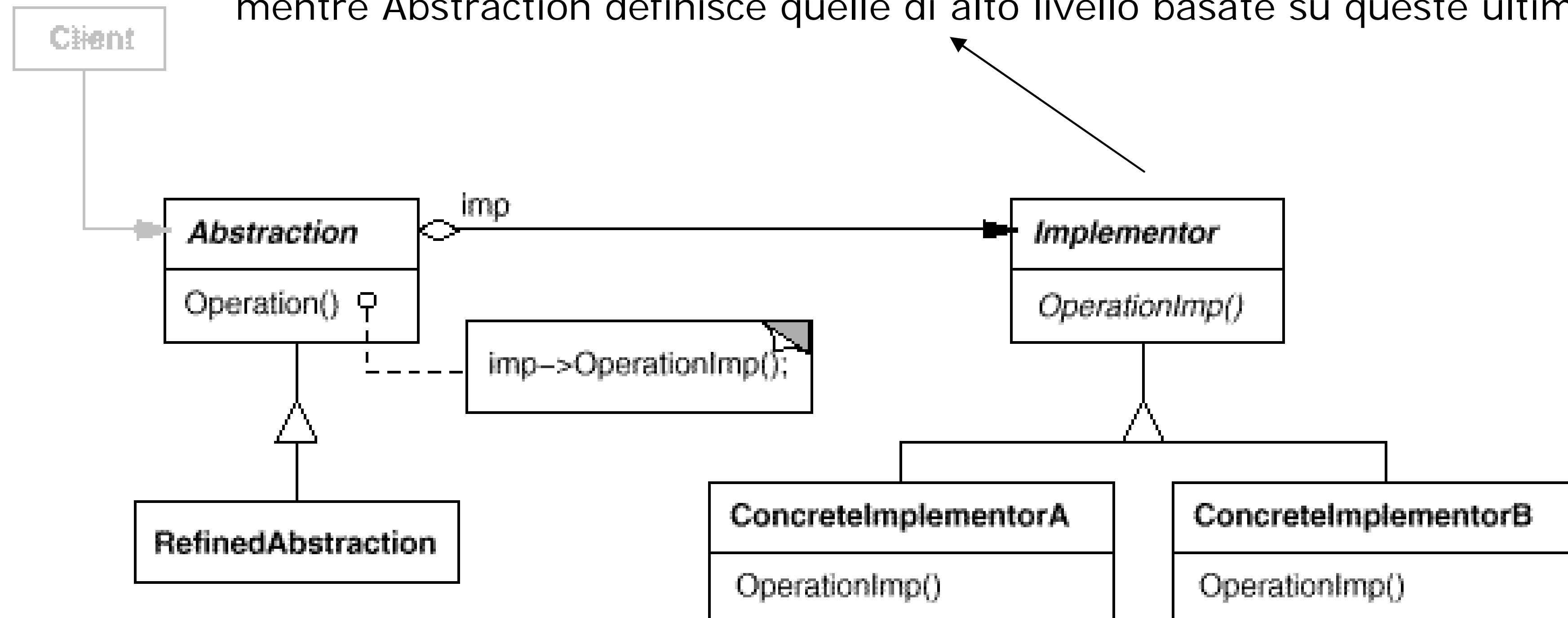
- Non è semplice estendere il codice per supportare una nuova tipologia di astrazione (v. slide precedente)
- Il codice del client è platform-dependent:
  ovunque un client richieda una astrazione, questi deve istanziare una classe concreta che ha una specifica implementazione
  (deve conoscere con quale implementazione si sta lavorando)
- un'astrazione è condizionata da un'implementazione in maniera *esplicita* e *statica*

Bridge (iii) (object/structural)

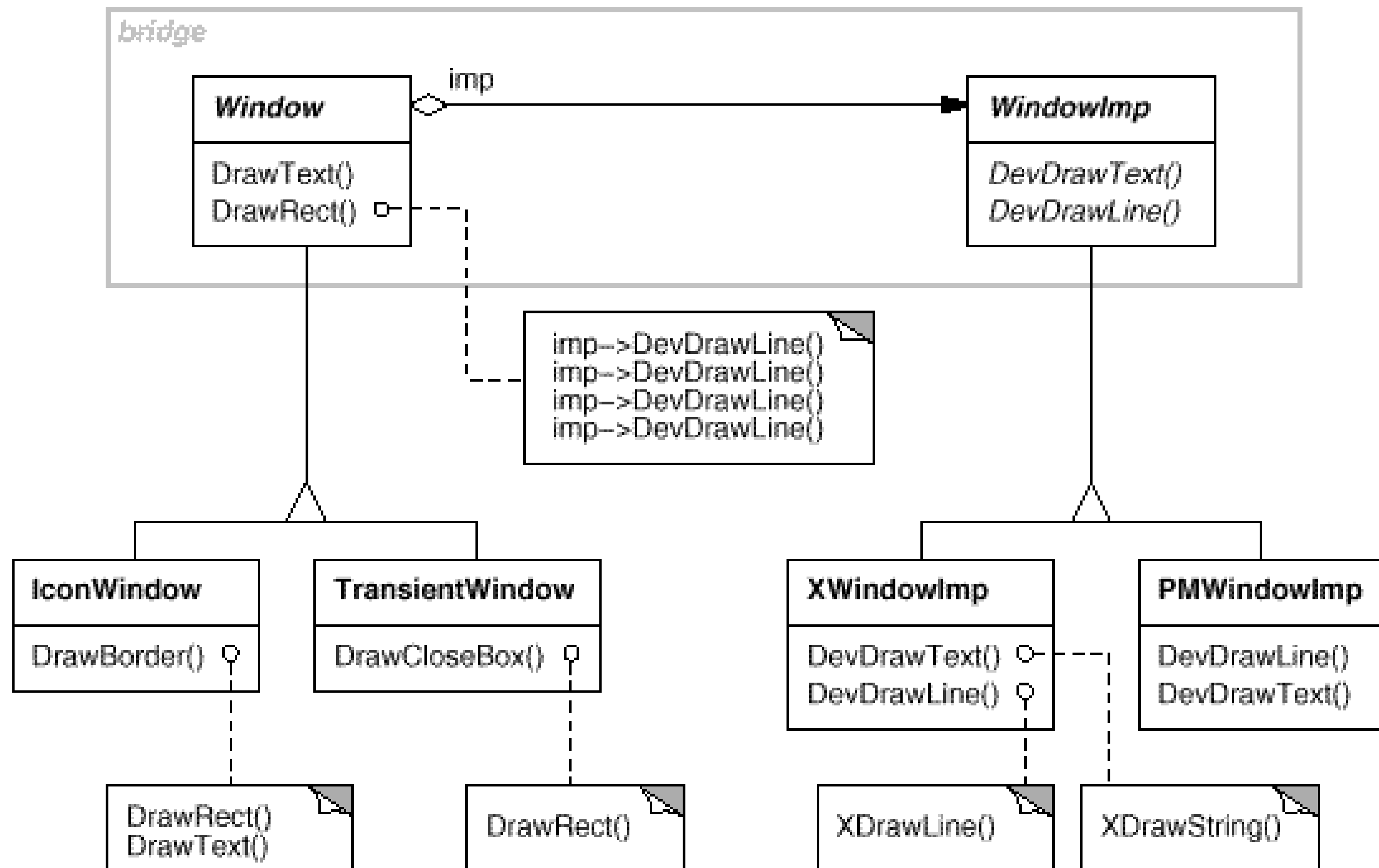


Bridge (iv)

Definisce l'interfaccia per le classi di implementazione. Questa interfaccia non deve necessariamente coincidere con l'interfaccia di *Abstraction*; anzi, tipicamente l'interfaccia di Implementor fornisce solo operazioni primitive, mentre Abstraction definisce quelle di alto livello basate su queste ultime.



Bridge – Esempio



Bridge – Codice

punta all'oggetto che implementa le funzioni di basso livello

```
class Window {
  public:
     Window(View* contents);

     // requests handled by window
     virtual void DrawContents();

     virtual void Open();
     virtual void Close();
     virtual void Iconify();
     virtual void Deiconify();

     // requests forwarded to implementation
     virtual void SetOrigin(const Point& at);
     virtual void SetExtent(const Point& extent);
     virtual void Raise();
     virtual void Lower();

     virtual void DrawLine(const Point&, const Point&);
     virtual void DrawRect(const Point&, const Point&);
     virtual void DrawPolygon(const Point[], int n);
     virtual void DrawText(const char*, const Point&);

  protected:
     WindowImp* GetWindowImp();
     View* GetView();

  private:
     WindowImp* _imp;
     View* _contents; // the window's contents
  } // end of class Window;


void Window::DrawRect (const Point& p1, const
      Point& p2) {
     WindowImp* imp = GetWindowImp();
     imp->DeviceRect(p1.X(), p1.Y(), p2.X(),
      p2.Y());
  }
```

in questo esempio le classi di tipo Window accedono all'implementazione attraverso una funzione di accesso

Bridge – Codice (ii)

ridefinizione di una funzione usando l'implementazione...

```
class IconWindow : public Window {
   public:
      // ...
      virtual void DrawContents();
   private:
      const char* _bitmapName;
   };

void IconWindow::DrawContents() {
      WindowImp* imp = GetWindowImp();
      if (imp != 0) {
         imp->DeviceBitmap(_bitmapName,
       0.0, 0.0);
      }
   }
```

```
class WindowImp {
   public:
      virtual void ImpTop() = 0;
      virtual void ImpBottom() = 0;
      virtual void ImpSetExtent(const Point&) = 0;
      virtual void ImpSetOrigin(const Point&) = 0;

      virtual void DeviceRect(Coord, Coord, Coord, Coord) = 0;
      virtual void DeviceText(const char*, Coord, Coord) = 0;
      virtual void DeviceBitmap(const char*, Coord, Coord) = 0;
      // lots more functions for drawing on windows...
   protected:
      WindowImp();
   };
```

Bridge – Codice (iii)

```
class XWindowImp : public WindowImp {
   public:
      // ...
virtual voidDeviceRect(Coord,Coord,Coord,Coord);
//...
private:
      //XWindow specific state
   };

void XWindowImp::DeviceRect(Coord x0, Coord
        y0, Coord x1, Coord y1 ) {
int x=round(min(x0,x1));
int y=round(min(x0,x1));
int w=round(abs(x0-x1));
int h=round(abs(y0-y1));
XDrawRectangle( /*....*/ , x,y,w,h);
//un rectangle è definito come l'angolo in basso a
//sinistra + width e height
//...
}
```

```
class PMWindowImp : public WindowImp {
   public:
      // ...
virtual voidDeviceRect(Coord,Coord,Coord,Coord);
//...
private:
      //PMWindow specific state
   };

void PMWindowImp::DeviceRect(Coord x0, Coord y0,
        Coord x1, Coord y1 ) {
Coord left=min(x0,x1);
Coord right=max(x0,x1);
Coord bottom=min(y0,y1);
Coord top=max(y0,y1);
//..costruzione dei 4 vertici -> PPOINTL point[4];
//..costruizione del path ->
GpiStrokePath(/*...*/);
//per es. in questo sistema grafico non esiste una
        //primitiva per disegnare rettangoli ma spezzate}
```

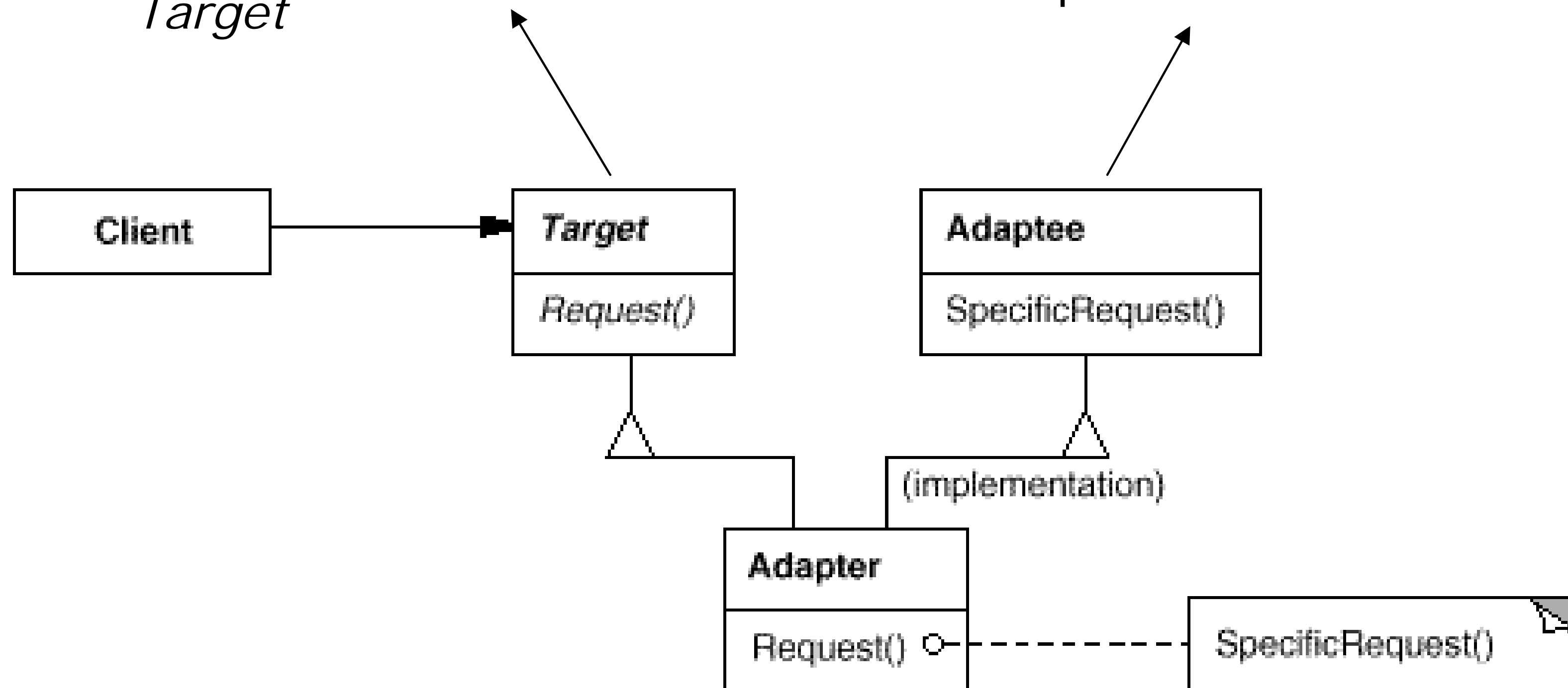
Adapter (class,object / structural)

- Converte l'interfaccia di una classe in un'altra interfaccia che i client si aspettano
- Adapter permette a due classi con interfacce incompatibili di cooperare

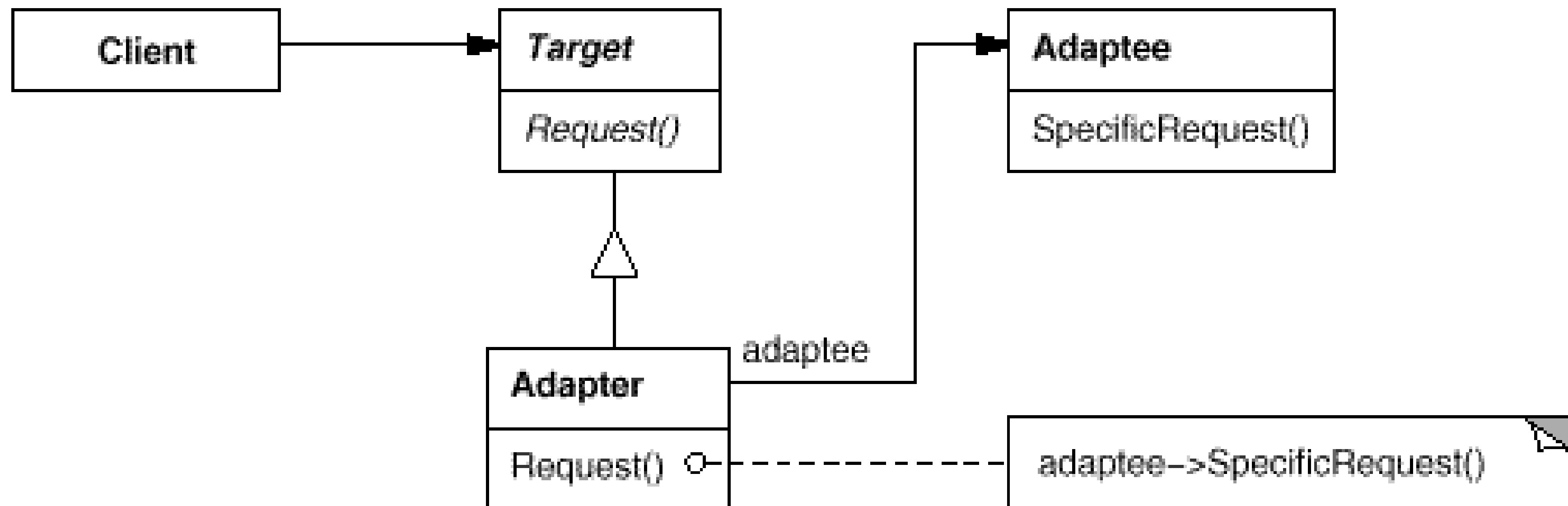
- Due diversi approcci:
  1. Basato su EREDITARIETA' MULTIPLA
  2. Basato su COMPOSIZIONE

- Spesso l'*Adapter* fornisce anche funzionalità che la classe adattata non è in grado di supportare

Adapter – Ereditarietà multipla



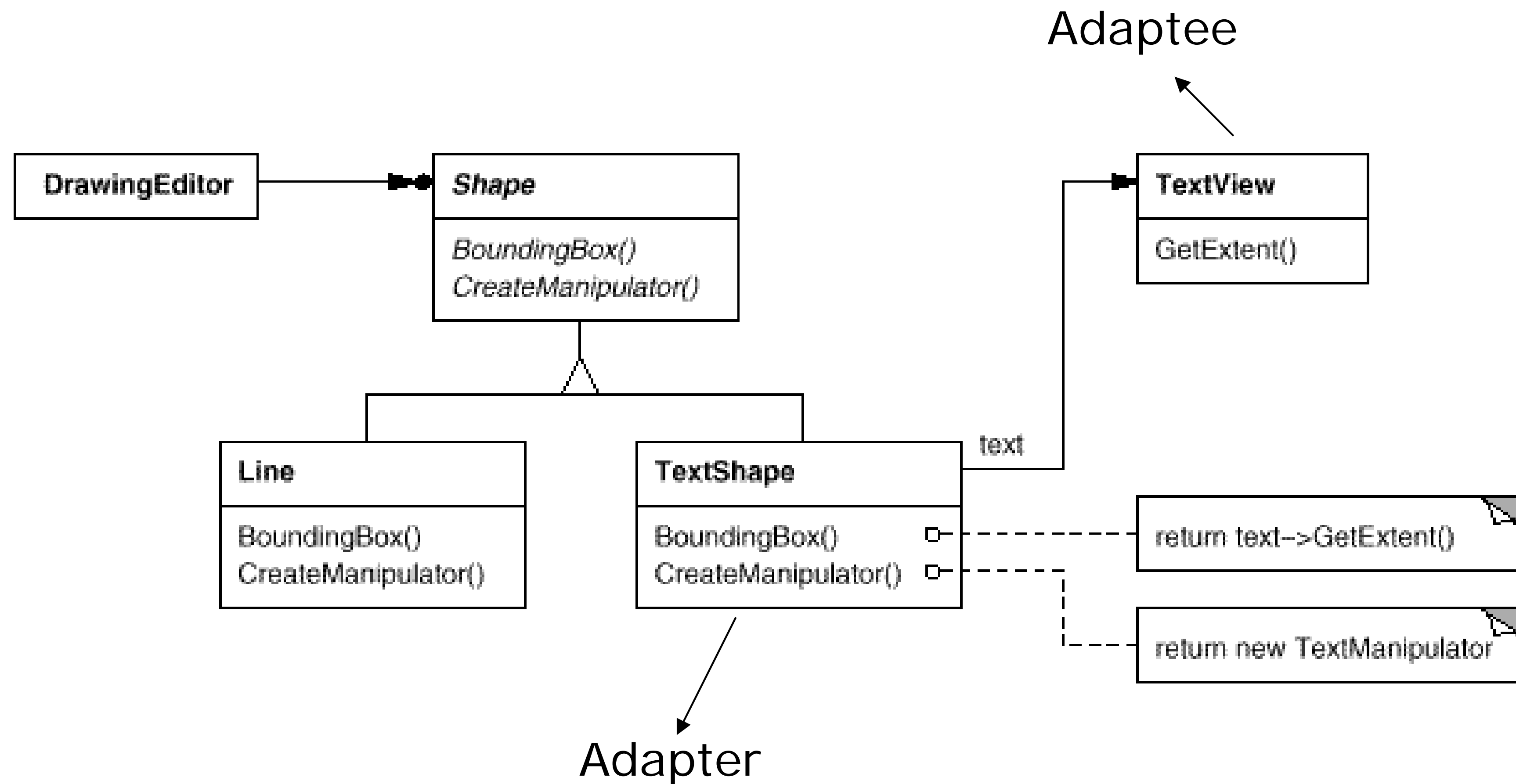
Adapter – Composizione



Adapter (ii)

- Approccio basato su **ereditarietà**:
  - L'interfaccia di Adaptee è adattata all'interfaccia Target appoggiandosi ad una classe Adaptee concreta
    - Di conseguenza, una classe Adapter non può essere utilizzata quando si vuole adattare una classe e tutte le sue sottoclassi
  - Consente alla classe Adapter di sovrascrivere parte del comportamento di Adaptee, visto che Adapter è una sottoclasse di Adaptee
  - Introduce soltanto un oggetto, e non occorrono ulteriori indirezioni per ottenere un riferimento all'oggetto adattato (design più semplice e intuitivo)
- Approccio basato su **composizione**:
  - Permette a un'unica classe Adapter di lavorare con Adaptee e tutte le sue sottoclassi
    - L'Adapter può anche aggiungere delle funzionalità a tutti gli Adaptee contemporaneamente
  - Introduce un oggetto ulteriore complicando il design

Adapter – Esempio



Manipulator: definisce oggetti per animare una “Shape”->drag and drop

Adapter – Codice

adaptee

```
class Shape{
public:
 Shape();
 virtual void BoundingBox(Point& bottomLeft, Point& topRigth) const;
virtual Manipulator* CreateManipulator()const;
}
```

```
class TextView{
public:
 TextView();
void GetOrigin(Coord&x, Coord&y)const;
void GetExtent(Coord&width,Coord&heigth)const;
virtual bool IsEmpty()const;
}
```

```
class TextShape:public Shape,private TextView{
public:
 TextShape();
 virtual void BoundingBox(Point& bottomLeft, Point& topRigth) const;
virtual Manipulator* CreateManipulator()const;
virtual bool IsEmpty()const;
}
```

eredita l'interfaccia di Shape e l'implementazione di TextView

adapter (di che tipo?)

Adapter – Codice

```
void TextShape::BoundingBox(Point& bottomLeft, Point& topRight)const{
Coord bottom,left,width,heigth;
GetOrigin(bottom,left);
GetExtent(width,height);
bottomLeft=Point(bottom,left);
topRigth=Point(bottom+heigth,left+width);
}
bool TextShape::IsEmpty()const{
return TextView::IsEmpty();
}
Manipulator* TextShape::CreateManipulator()const{
return new TextManipulator(this);
}
```

→ implementazione dell'interfaccia di Shape utilizzando l'interfaccia di TextView

→ inoltro diretto di una richiesta all'adaptee

→ implementazione di una operazione dell'interfaccia di Shape che non usa nessuna delle funzionalità di TextView

Adapter – Codice: object comp.

```
class TextShape :: public Shape {
public:
  TextShape(TextView*);
  virtual void BoundingBox(Point& bottomLeft,Point& topRigth) const;
  virtual bool IsEmpty ()const;
  virtual Manipulator* CreateManipulator()const;
private:
  TextView* _text;
}

TextShape::TextShape(TextView* t){
_text=t;
}
```

maggiore overhead

maggiore flessibilità

Adapter – Codice: object comp.

```
void TextShape::BoundingBox(Point& bottomLeft, Point& topRight)const{
Coord bottom,left,width,heigth;
_text->GetOrigin(bottom,left);
_text->GetExtent(width,height);
bottomLeft=Point(bottom,left);
topRigth=Point(bottom+heigth,left+width);
}
```

→ implementazione dell'interfaccia di Shape utilizzando l'interfaccia di TextView

```
bool TextShape::IsEmpty()const{
return _text->IsEmpty();
}
```

→ inoltro diretto di una richiesta all'adaptee

```
Manipulator* TextShape::CreateManipulator()const{
return new TextManipulator(this);
}
```

→ in questo caso non è cambiato nulla: l'operazione non utilizza l'interfaccia di TextView

Iterator (object / behavioral)

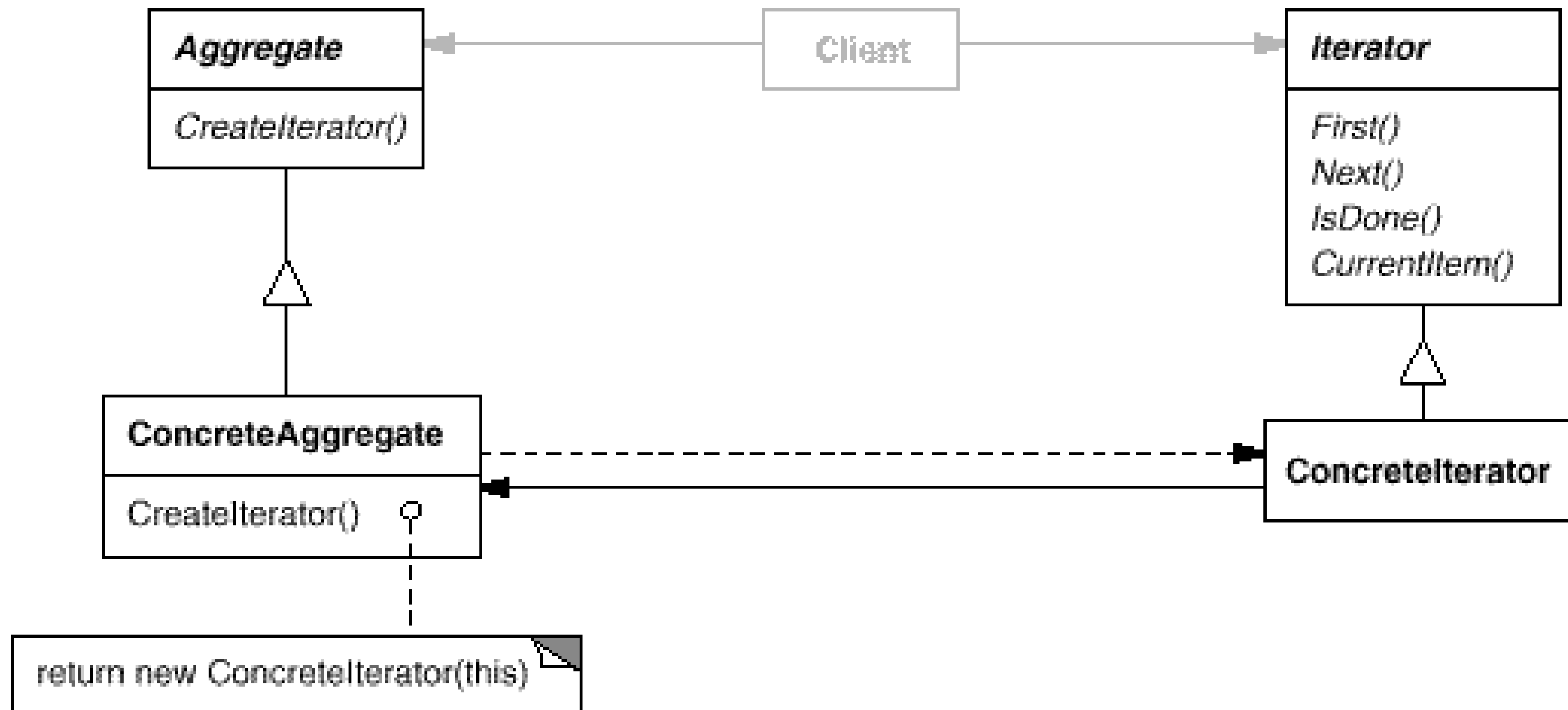
- Fornisce un modo per accedere agli elementi di un oggetto aggregato in maniera **sequenziale**, senza esportarne la rappresentazione interna

- Idea: togliere le funzionalità di accesso e attraversamento dall'aggregato e inserirle all'interno di un oggetto **iterator**

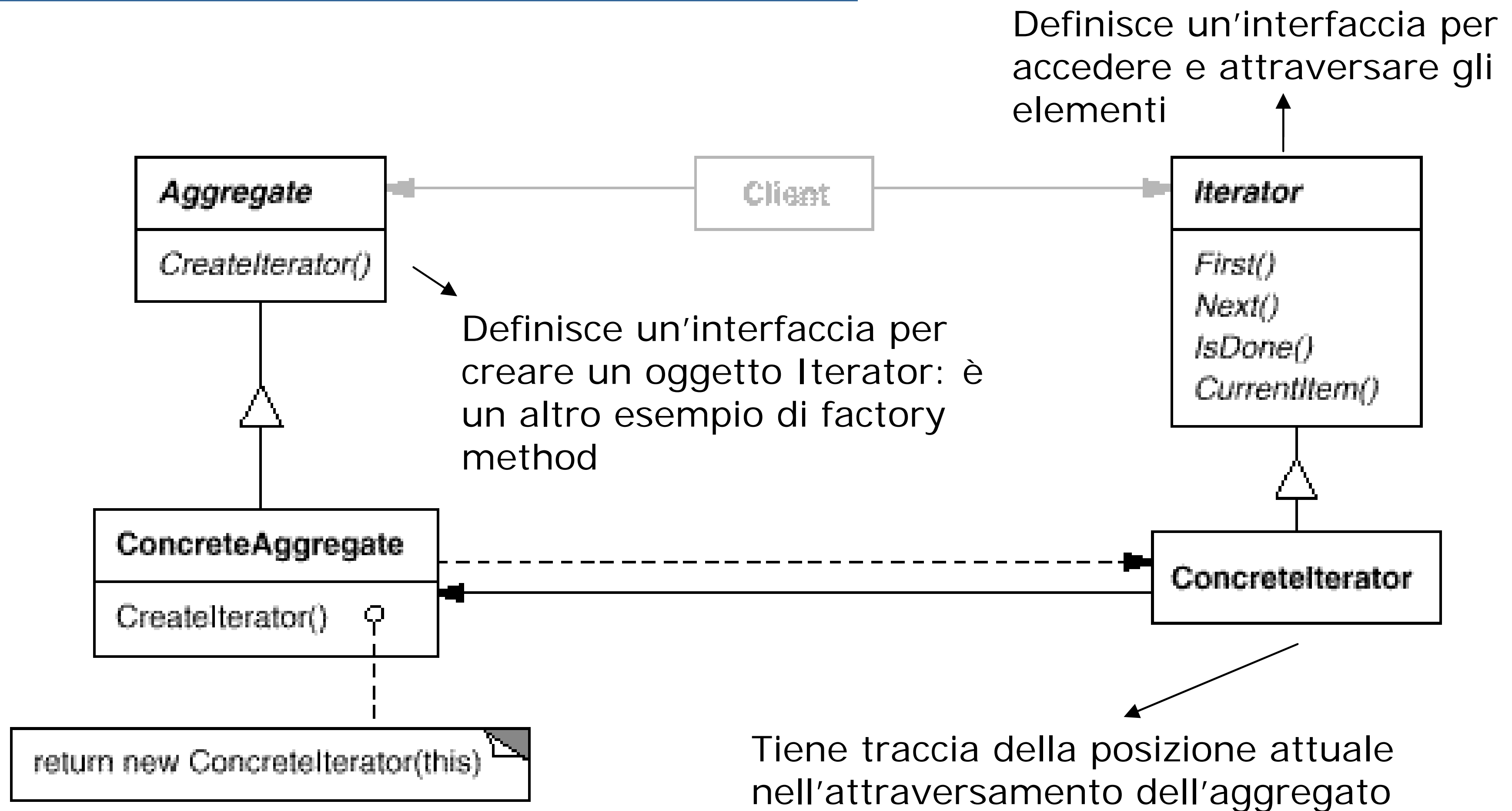
- Un oggetto iterator è responsabile di tener traccia dell'elemento corrente
  - Conosce quali elementi sono già stati attraversati
  - E' possibile utilizzare contemporaneamente più iteratori sulla stessa struttura dati

- Per es., data una lista, l'iterator:
  - Fornisce l'accesso ai suoi elementi senza esportare la sua interfaccia
  - Evita di appesantire l'interfaccia della lista con operazioni per i diversi tipi di attraversamento (ad. es. reverse o filtrato)
  - Svincola il codice del client dall'implementazione della str. dati

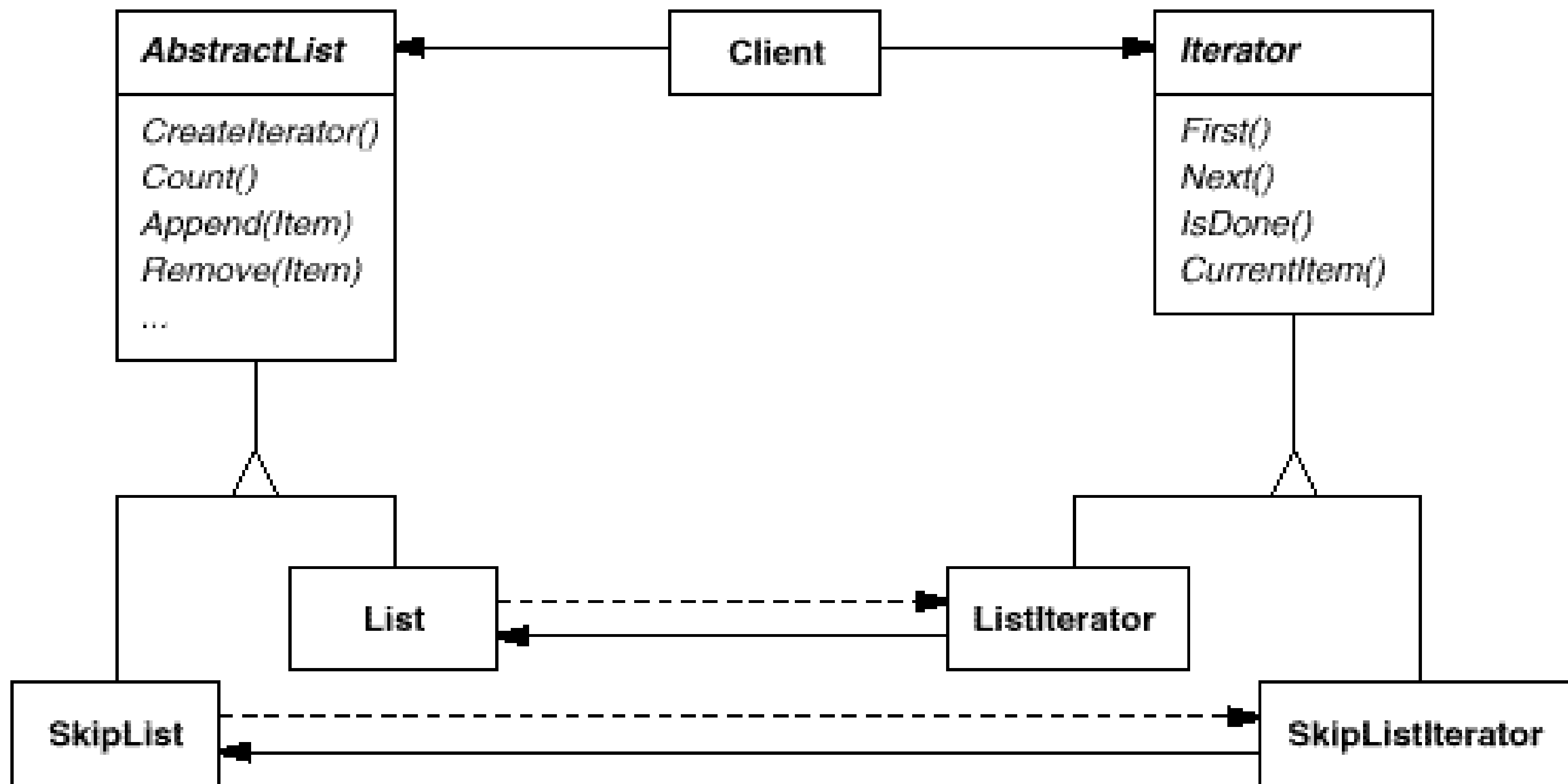
Iterator (ii)



Iterator (iii)



Iterator – Esempio



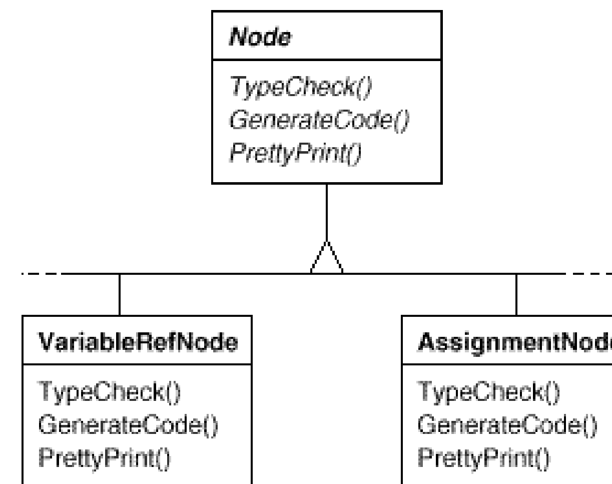
Iterator (iv)

- E' possibile definire più *concrete iterator* che visitano lo stesso contenitore con politiche diverse
  - Approccio adatto soprattutto per gli alberi

- Ogni iterator deve conoscere i dettagli della classe che visita
- La STL del linguaggio C++ fa ampio uso del pattern iterator

Visitor

- Requisiti:
  - Ho un aggregato (per es. un albero) i cui elementi sono, per natura, eterogenei tra loro
  - Globalmente l'aggregato deve supportare certe operazioni, anche molto diverse tra loro -> i singoli nodi devono supportare tali operazioni, che hanno effetto sullo specifico elemento
- Es.: compilatori e abstract syntax trees

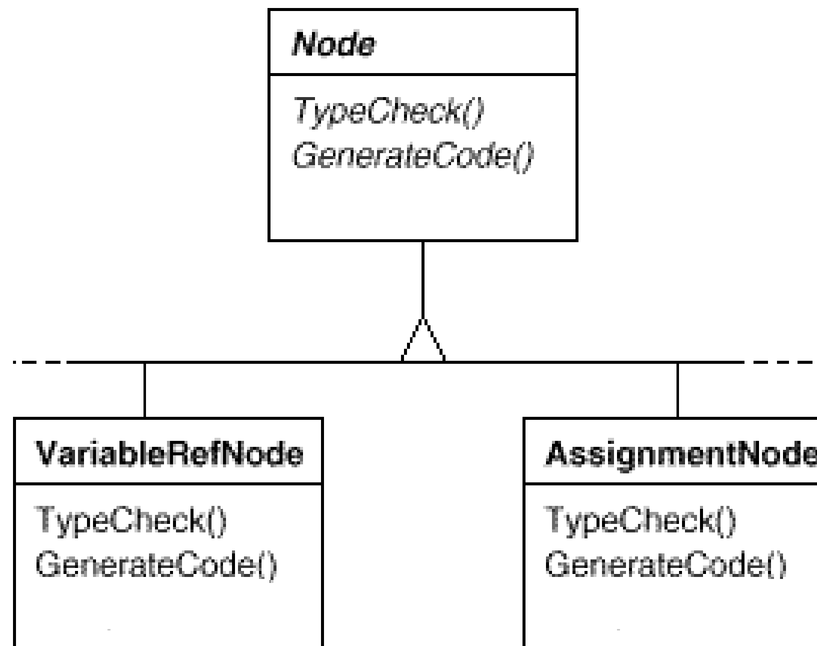


Visitor (ii)

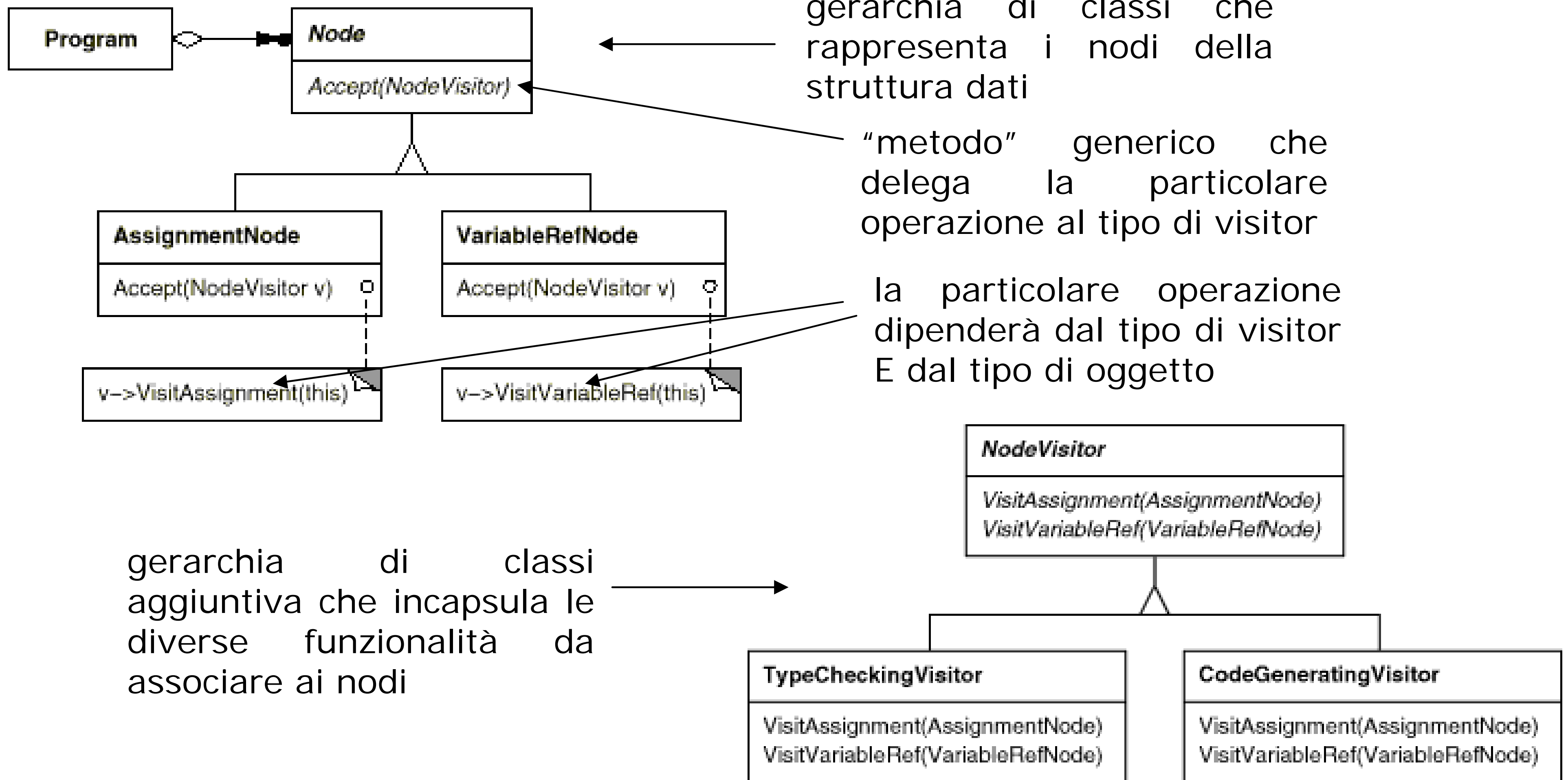
- Un Visitor rappresenta un'operazione che deve essere operata sugli elementi di un aggregato (elementi di tipo diverso tra di loro)

- Visitor permette di definire una <u>nuova</u> operazione <u>senza cambiare le classi degli elementi su cui questa opera</u>

Visitor – Esempio



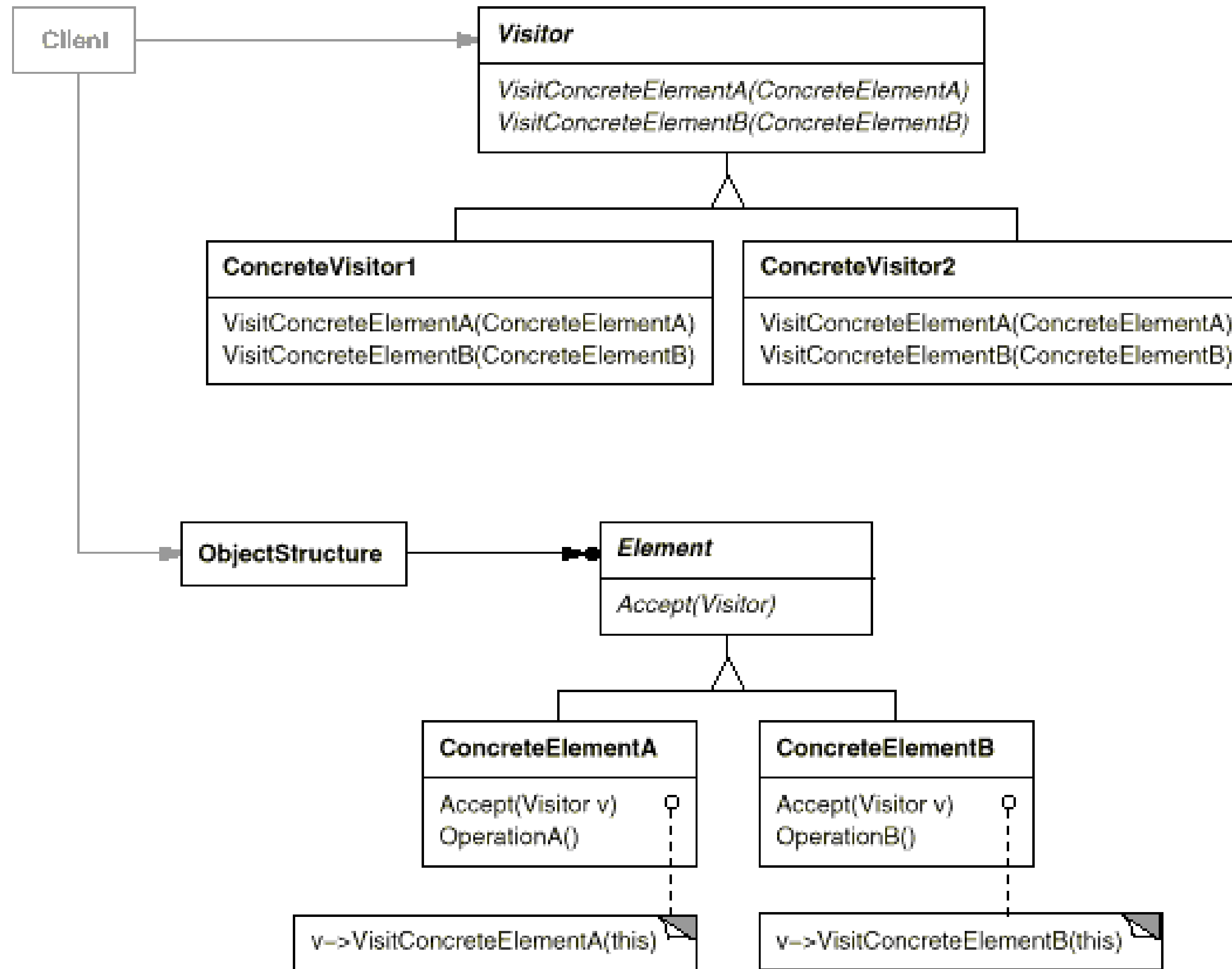
Visitor – Esempio (ii)



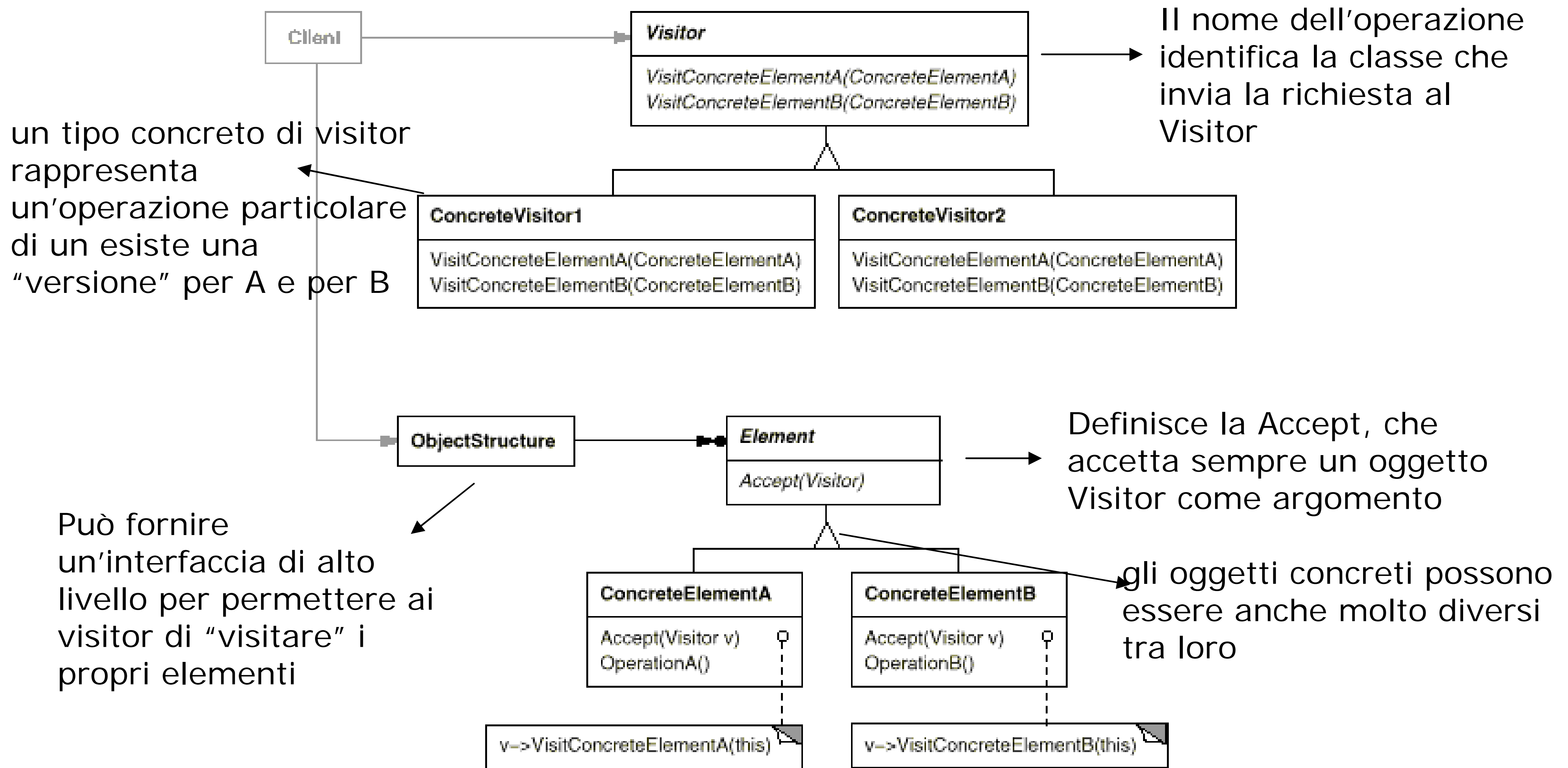
Visitor (iii) quando utilizzarlo

- una struttura di oggetti (ad esempio un aggregato ...) contiene molte classi con interfacce diverse e le operazioni da invocare sugli oggetti dipendono dal loro tipo
- le varie operazioni di un oggetto sono diverse, distinte e scorrelate tra di loro->invece che appesantirne l'interfaccia si isolano funzionalità simili in un Visitor appropriato
- le classi che compongono la struttura degli oggetti non cambiano frequentemente, mentre frequentemente c'è bisogno di definire nuovi tipi di operazioni (...)

Visitor (object/structural)

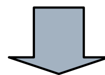


Visitor (iv)



Proxy

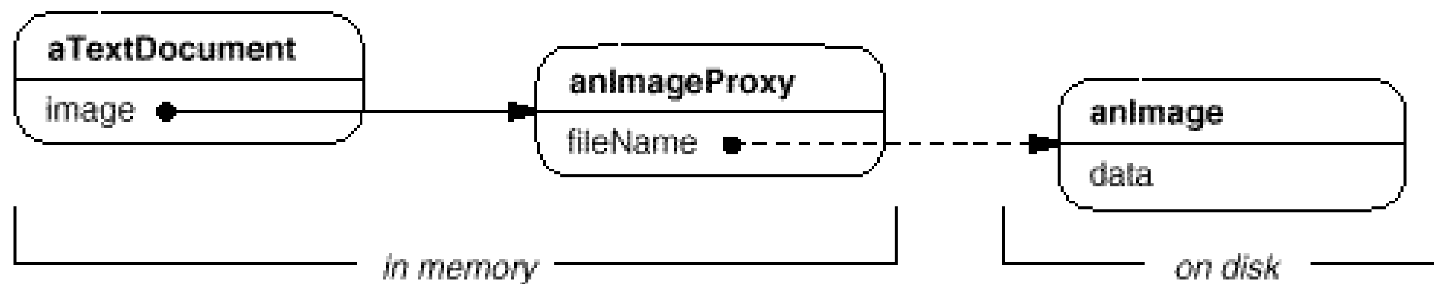
- Fornisce un “surrogato” di un oggetto e ne controlla gli accessi
- A cosa serve controllare gli accessi ad un oggetto?
  - Ritardare il costo totale della sua creazione e inizializzazione fino a che non sia effettivamente necessario
- Es.: editor di testo che può inglobare delle immagini nel documento
  - La visualizzazione di alcune immagini può essere costosa MA ...
  - Aprire il documento deve avvenire velocemente



  - Evitare di creare tutte le immagini contemporaneamente all’apertura del documento (non tutte le immagini sono visibili allo stesso tempo) -> creazione *on-demand*

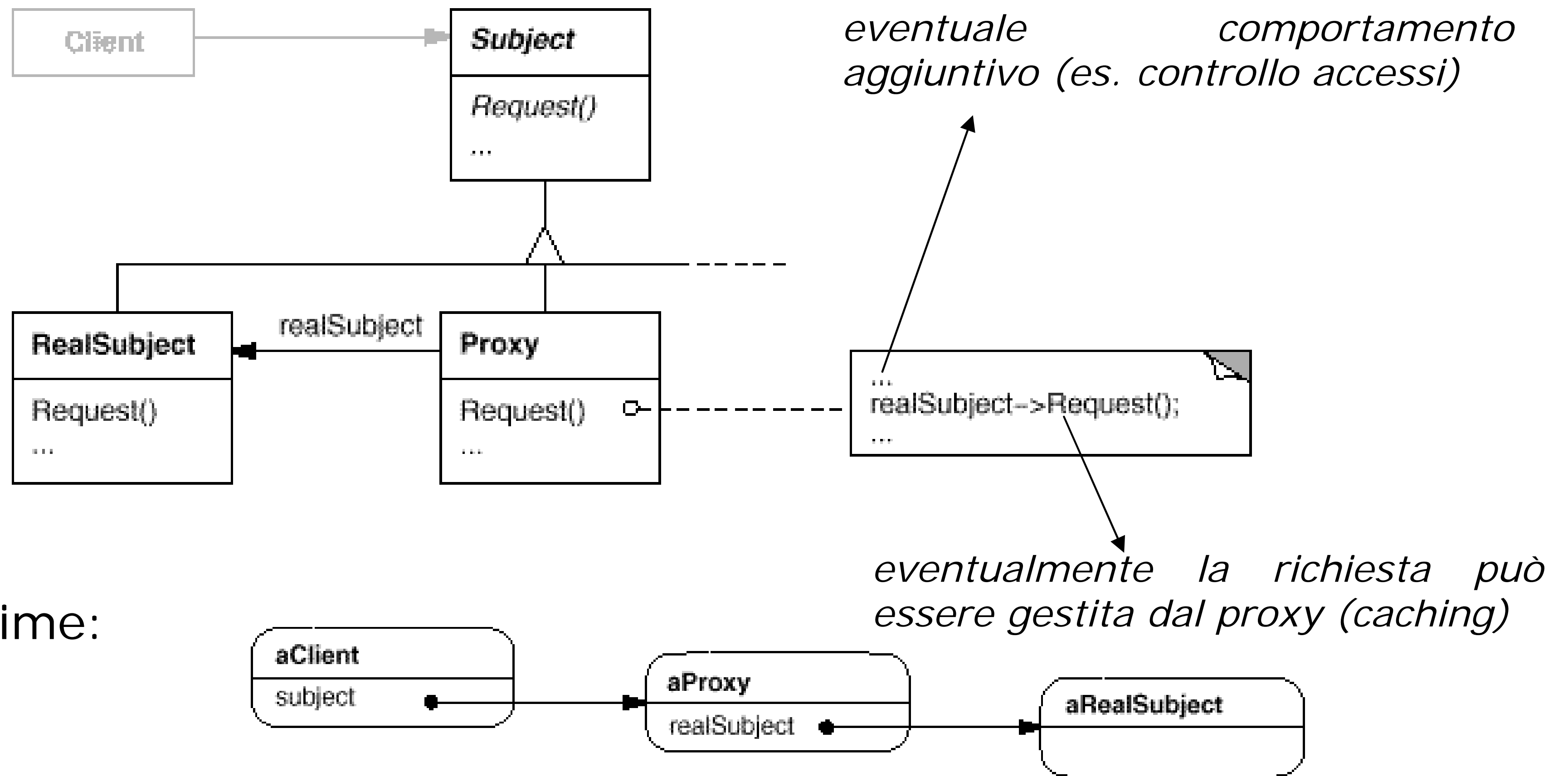
Proxy (ii)

- Soluzione: oggetto *Proxy* che agisce come segnaposto dell'oggetto concreto



- Il proxy dell'immagine carica l'immagine dal disco solo quando l'editor di testo gli chiede di visualizzarla (invocando l'operazione Draw())
- Il proxy inoltra le richieste successive direttamente all'immagine

Proxy (iii) (object/structural)



Proxy – Esempio

Un editor di testo (*DocumentEditor*) può gestire elementi di tipo *Graphic**, che supportano le operazioni:

Draw()
GetExtent() /* per sapere le dimensioni – lunghezza e larghezza */
Store() / ... /
Load()

Un elemento grafico può essere un'immagine, che può essere creata direttamente o *on-demand*, attraverso un **proxy**.

Si assume che le immagini siano immagazzinate in file separati e che si possa usare il nome del file come riferimento all'immagine su disco.

Si assume inoltre che l'oggetto proxy mantenga (come l'oggetto relativo all'immagine reale) un campo *extent* per contenere le dimensioni dell'immagine.

**Graphic* è l'interfaccia alla quale sia il proxy che l'immagine devono conformarsi

Proxy – Esempio (codice)

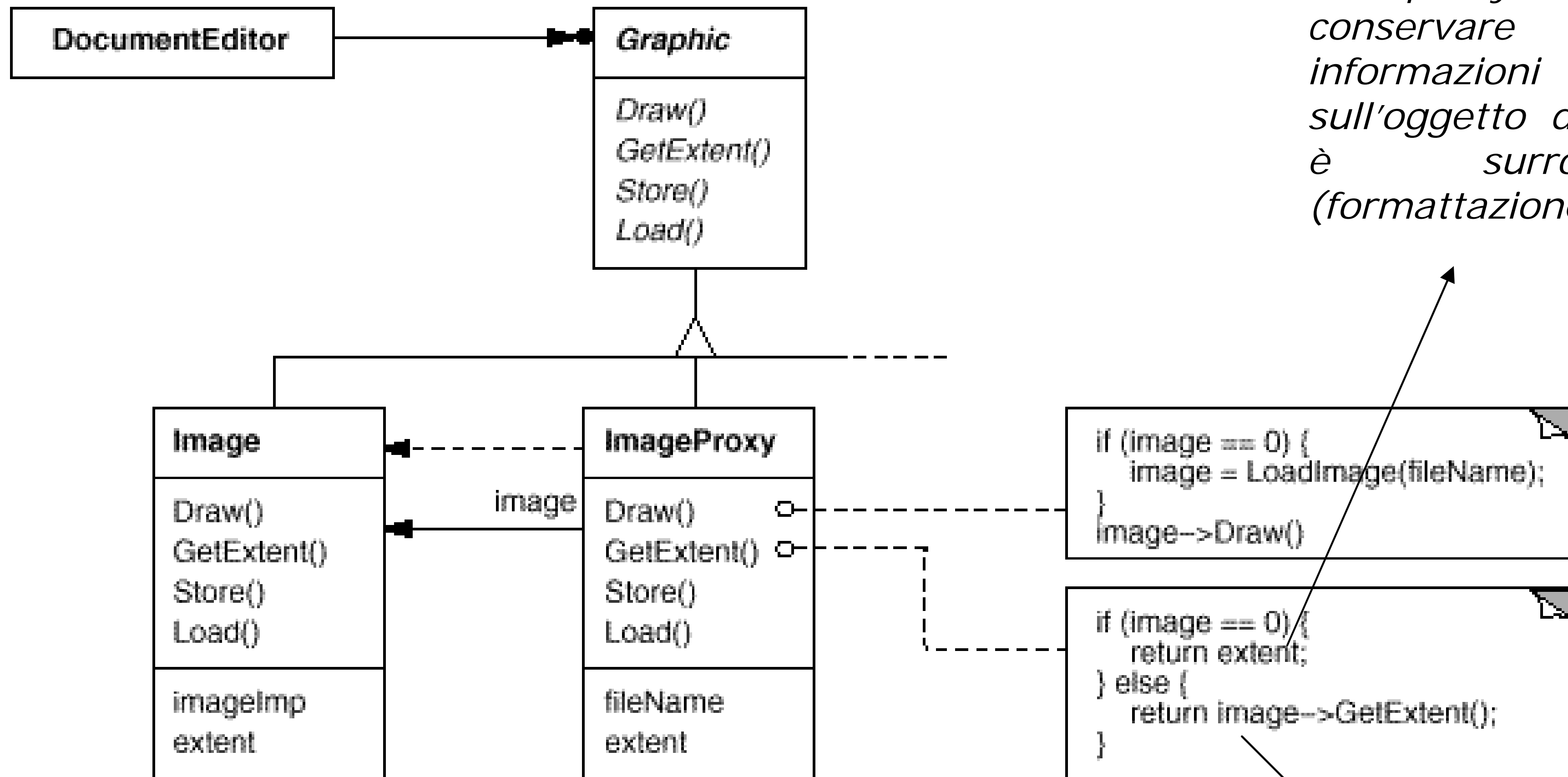
```
class Image : public Graphic {
public:
Image(const char* file); // loads
//image from a file
virtual ~Image();
virtual void Draw(const Point& at);
virtual void HandleMouse(Event&
event);
virtual const Point& GetExtent();
virtual void Load(istream& from);
virtual void Save(ostream& to);
private:
// ...
};
```

```
class ImageProxy : public Graphic {
public:
ImageProxy(const char* imageFile);
virtual ~ImageProxy();
virtual void Draw(const Point& at);
virtual void HandleMouse(Event& event);
virtual const Point& GetExtent();
virtual void Load(istream& from);
virtual void Save(ostream& to);
protected:
Image* GetImage();
private:
Image* _image;
Point _extent;
char* _fileName;
};
```

Proxy – Esempio (codice)

Proxy – Esempio

il proxy può conservare informazioni sull'oggetto di cui è surrogato (formattazione...)



si può migliorare?

Proxy – Utilizzi

- **virtual proxy** (creazione on demand)
  - istanziare subito tutti gli oggetti può essere costoso
  - caching->il proxy può soddisfare alcune richieste senza interrogare l'oggetto di cui è *surrogato*
- **remote proxy** (*sostituto* locale di un oggetto remoto in un sistema distribuito, ad esempio in un altro calcolatore connesso in rete)
  - gestire la comunicazione (*trasparenza*)
  - ancora->caching
- **protection proxy** (a cui delego una politica di accesso per una risorsa)

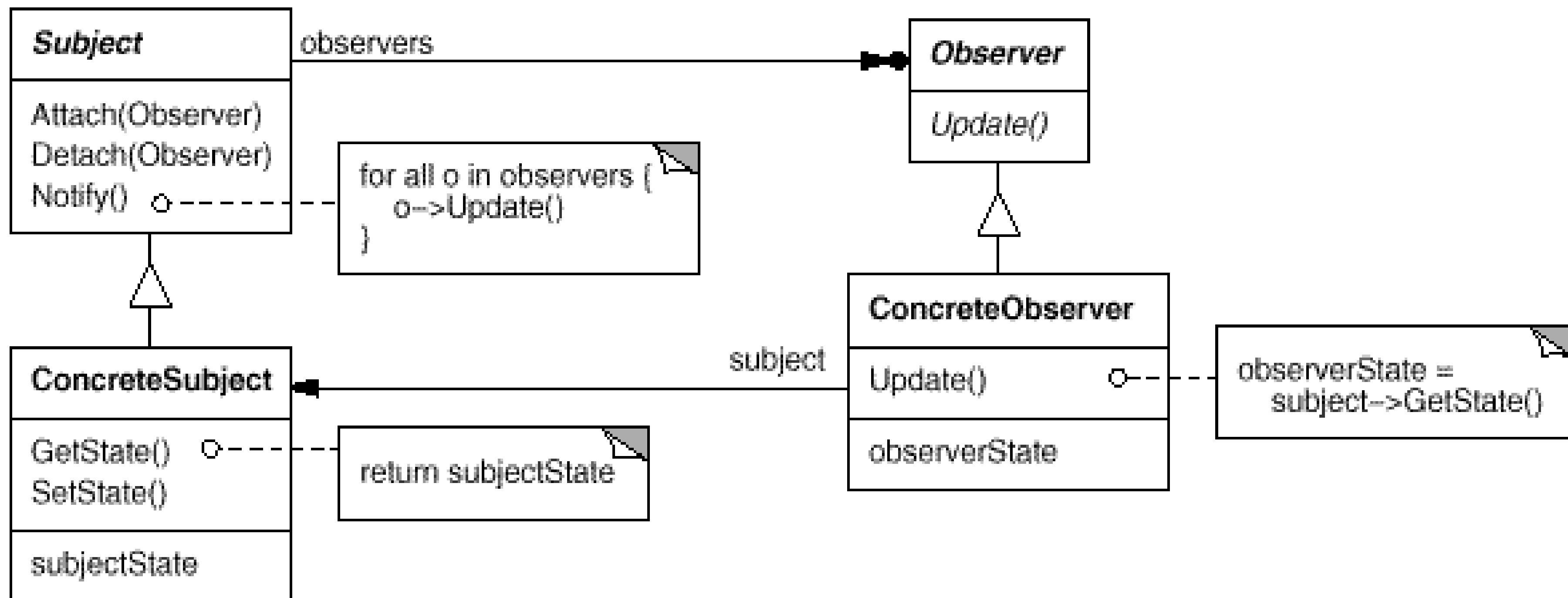
- **smart reference (...)**

Observer (object/behavioral)

- Definisce una dipendenza tra oggetti di tipo *uno-a-molti*: quando lo stato di un oggetto cambia, tale evento viene notificato a tutti gli oggetti dipendenti: essi vengono automaticamente aggiornati.

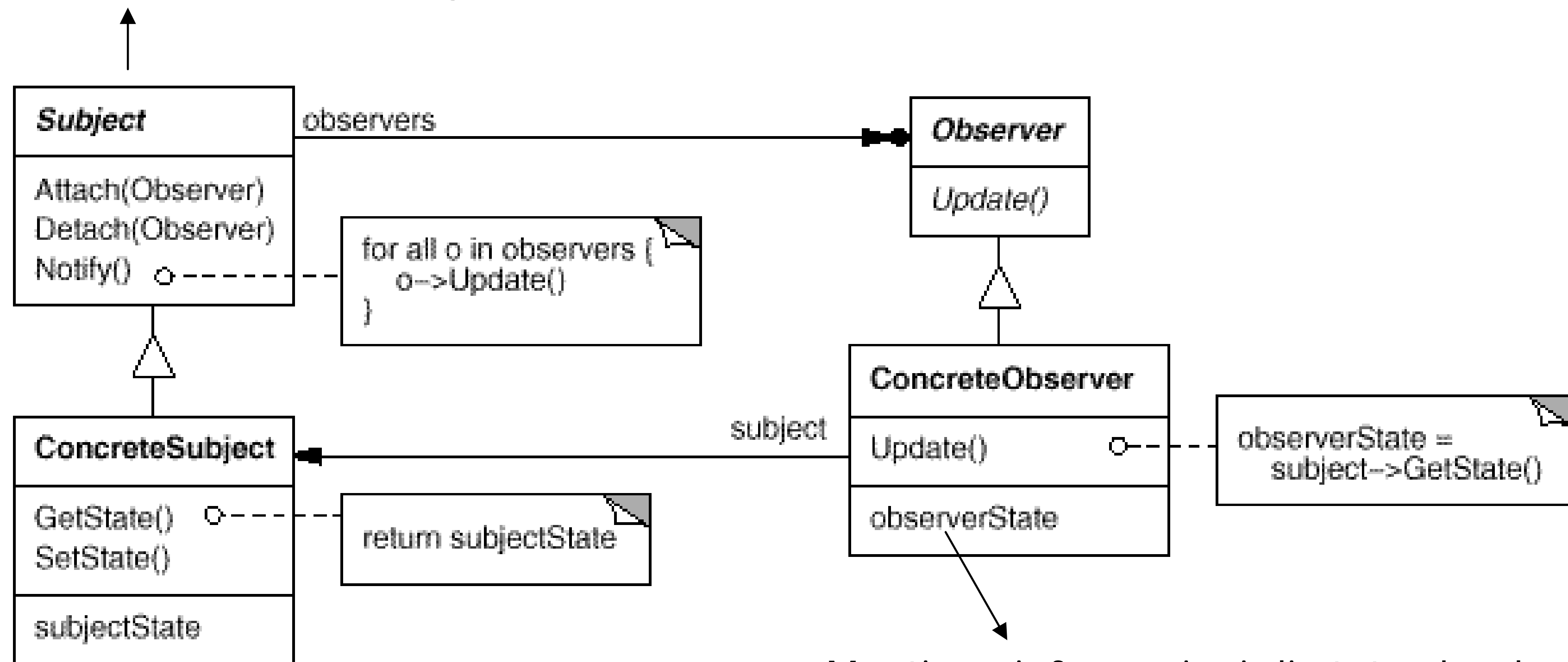
- **Contesto**
  - Uno o più oggetti sono interessati ad *osservare* i cambiamenti di stato di un soggetto
- **Problema**
  - Il soggetto deve essere indipendente dal numero e dal tipo degli osservatori
  - Deve essere possibile aggiungere(rimuovere) *osservatori* durante l'esecuzione dell'applicazione
- **Soluzione**
  - Inserire nel soggetto delle operazioni che consentano all'osservatore di dichiarare il proprio interesse per un cambiamento di stato (attach/detach)

Observer (ii)



Observer (iii)

Fornisce un'interfaccia per "aggiungere" e "rimuovere" un un Observer dalla lista degli Observer da notificare



Mantiene informazioni di stato che devono essere consistenti con lo stato del soggetto

Observer (iii)

Quando applicare il pattern Observer:

- Quando un cambiamento (di stato) di un oggetto richiede il cambiamento (di stato) di altri oggetti, e il loro numero non è noto a priori
- Quando un oggetto deve essere in grado di notificare cambiamenti di stato ad altri oggetti senza fare alcuna assunzione sulla natura di questi oggetti (il loro tipo)
  - Si vuole, cioè, che le due tipologie di oggetti (i soggetti e gli osservatori) siano *disaccoppiate*
- I JavaBeans sono componenti riusabili (per cui esistono appositi tool) implementati sulla base di questo pattern

Possibili estensioni:

- più di un oggetto Osservato (gli observer devono "*sapere*" da chi arriva l'aggiornamento)
- tipologie di comunicazione tra Observer e Subject
- per ridurre l'overhead gli Observer possono registrarsi solo per un particolare tipo di eventi

Observer – Codice (iv)

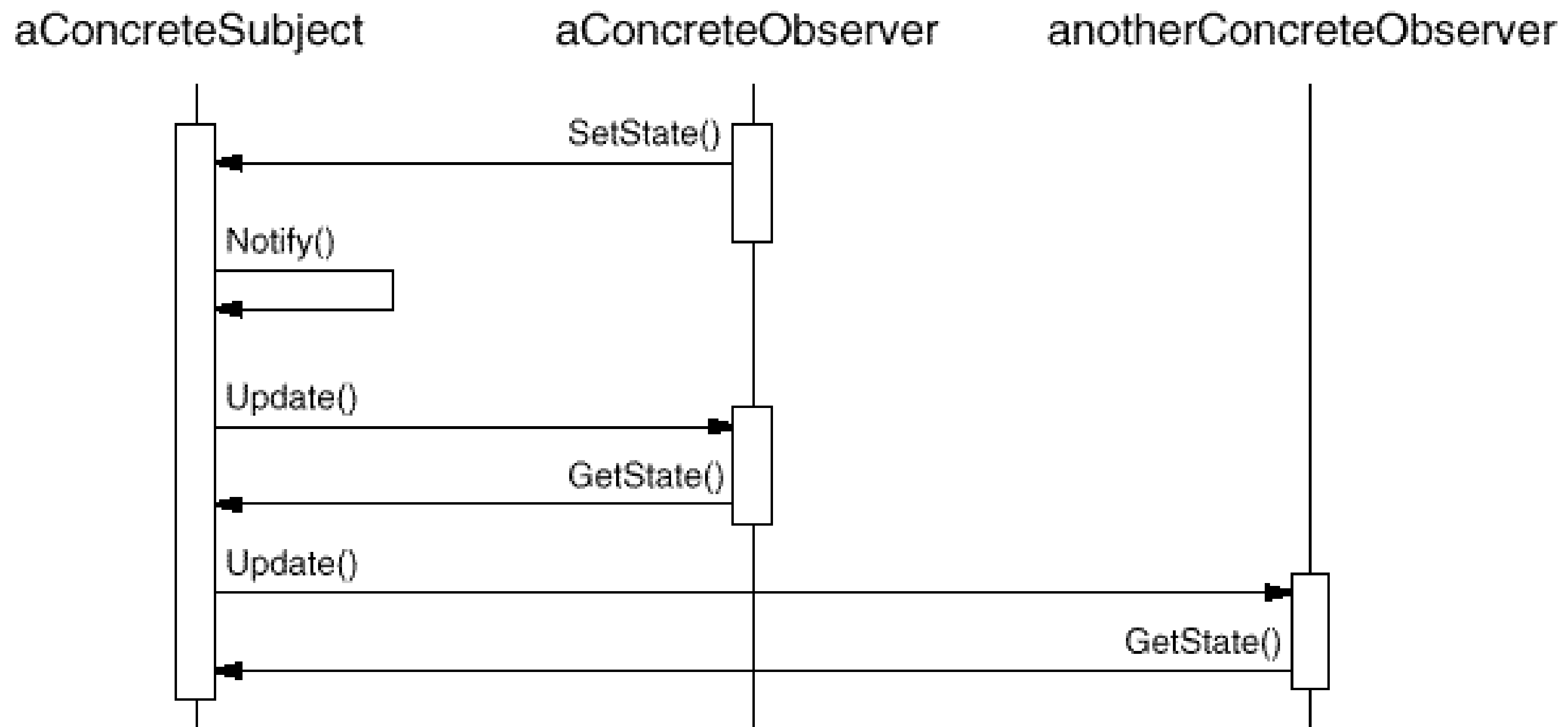
a cosa serve? (arrow pointing to `virtual ~ Observer();`)

```
class Observer {
public:
      virtual ~ Observer();
      virtual void Update(Subject*
      theChangedSubject) = 0;
protected:
      Observer();
};

class Subject {
public:
      virtual ~Subject();
      virtual void Attach(Observer*);
      virtual void Detach(Observer*);
      virtual void Notify();
protected:
      Subject();
private:
      List<Observer*> *_observers;
};
```

```
void Subject::Attach (Observer* o) {
      _observers->Append(o);
}

void Subject::Detach (Observer* o) {
      _observers->Remove(o);
}

void Subject::Notify () {
      ListIterator<Observer*>
      i(_observers);

      for (i.First(); !i.IsDone(); i.Next())
      {
        i.CurrentItem()->Update(this);
      }
}
```

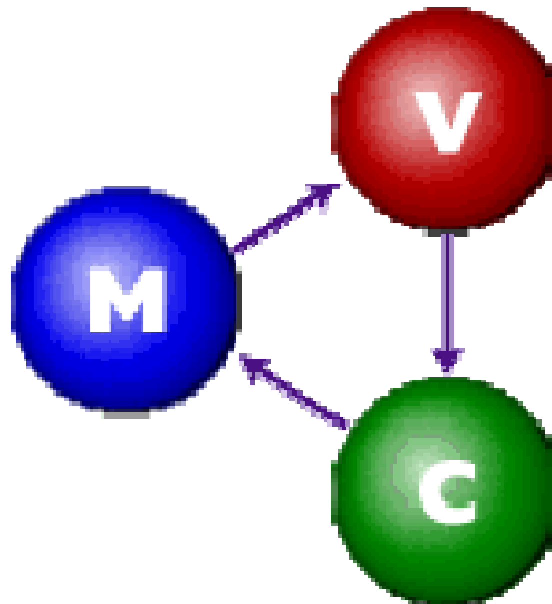
Observer (v)

Comportamento dinamico (diagramma di sequenza):



Model View Controller

Model*: rappresenta i dati dell’applicazione*



View*: rappresentazione visuale dei dati*

Controller*: gestisce l’input modificando View e Model*

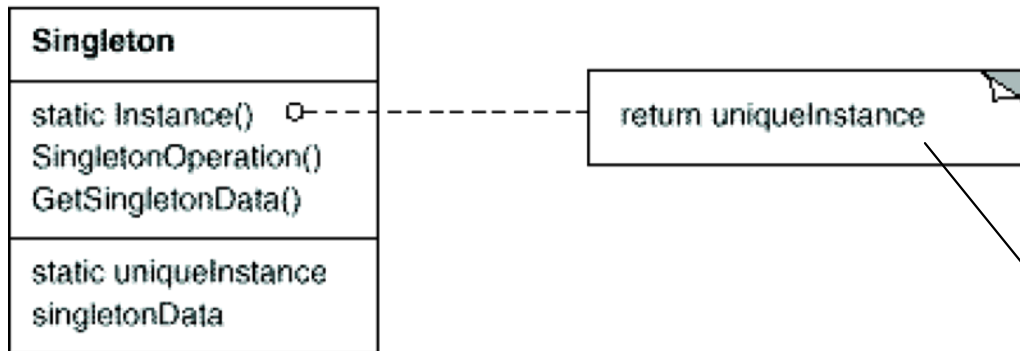
- introdotto con SmallTalk-80
- ha ispirato diversi framework grafici (ad es. il toolkit Java Swing)

Singleton (object/creational)

- Garantisce che una classe abbia un'unica istanza durante l'esecuzione del programma e ne offre un meccanismo globale di accesso. ("initialization on first use" / "lazy initialization")

- **Contesto**
  - Si vuole che il codice client di un oggetto sia sollevato dal dover creare esplicitamente oggetti di un tipo particolare e (chiamandone i costruttori) e gestirne (limitarne) il numero di istanze generate.
- **Ad esempio**
  - Gestore di connessioni ad un DB, window manager, factory, generatore di ID univoci, printer spooler. Lasciare che il codice client istanzi più oggetti di questi tipi può essere inefficiente o potenzialmente errato
- **Soluzione**
  - Il pattern Singleton rende la classe dell'oggetto stesso responsabile per l'eventuale creazione dell'istanza usando un metodo (statico) apposito a cui si delega la creazione e l'accesso controllato all'oggetto.

Singleton (ii)




```
class Singleton {
public:
	static Singleton* Instance();
	SingletonOperation();
	GetSingletonData();
protected:
	Singleton();
private:
	static Singleton* uniqueinstance;
	/*....*/
}
```

```
Singleton* Singleton::uniqueinstance=0;

Singleton* Singleton::Instance (){
	if(uniqueinstance==0){
		uniqueinstance=new Singleton();
	}
	return uniqueinstance;
}
```

Singleton (iii)

```
class IntegerSingleton
{
int m_value;
public:
IntegerSingleton ( int v=0 ) { m_value = v; }
int get_value() { return m_value; }
void set_value( int v ) { m_value = v; }
};
IntegerSingleton * global_ptr = 0;

void foo( void ) {
if ( ! global_ptr ) global_ptr = new IntegerSingleton ;
global_ptr->set_value( 1 );
cout << "foo: value is " << global_ptr->get_value() << '\n';
}
 void bar( void ) {
if ( ! global_ptr ) global_ptr = new IntegerSingleton ;
global_ptr->set_value( 2 );
cout << "bar: value is " << global_ptr->get_value() << '\n';
}
```

```
int main( void ) {
if ( ! global_ptr )
global_ptr = new IntegerSingleton ;
cout << "main: value is " << global_ptr->get_value() << '\n';
foo();
bar();
}
// main: global_ptr is 0
// foo: global_ptr is 1
// bar: global_ptr is 2
```

http://www.vincehuston.org/dp/singleton.html

Singleton (iv)

```
class IntegerSingleton
{
int m_value;
static IntegerSingleton* _instance;
IntegerSingleton ( int v=0 ) { m_value = v; }
public:
//IntegerSingleton ( int v=0 ) { m_value = v; }
int get_value() { return m_value; }
void set_value( int v ) { m_value = v; }
static IntegerSingleton* getInstance(){
        if(_instance==0)
                _instance=new IntegerSingleton;
        return _instance; }
};
IntegerSingleton* IntegerSingleton::_instance =0;

void foo( void ) {
IntegerSingleton::getInstance()->set_value( 1 );
cout << "foo: value is " <<
        IntegerSingleton::getInstance()-> get_value() << '\n';
}
```

```
void bar( void ) {
IntegerSingleton::getInstance()->set_value( 2 );
cout << "bar: value is " <<
IntegerSingleton::getInstance()-> get_value() << '\n';
}


int main( void ) {
cout << "main: value is " <<
        IntegerSingleton::getInstance()->get_value() << '\n';
foo();
bar();
}
// main: global_ptr is 0
// foo: global_ptr is 1
// bar: global_ptr is 2
```

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