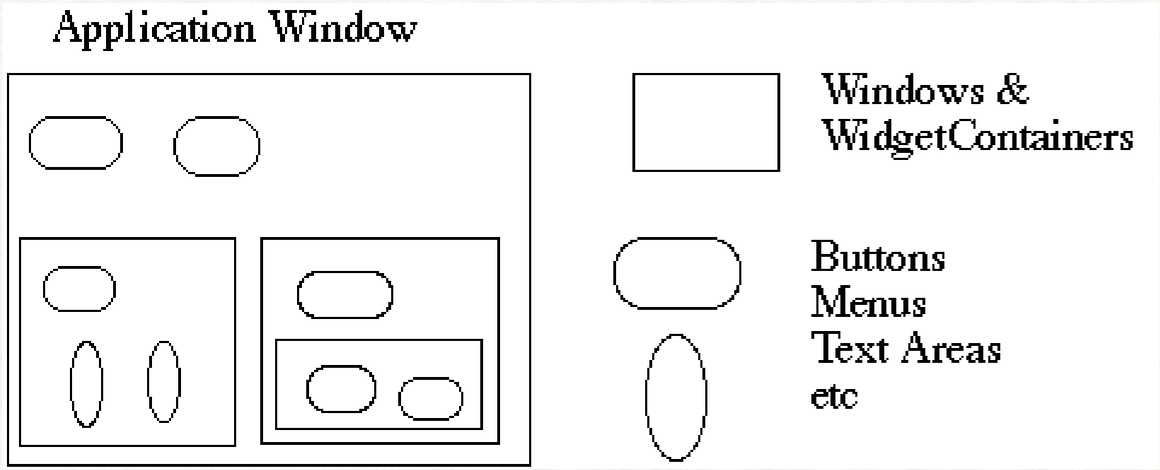
COMPOSITE

- Motivazione: esempio



- Come far sì che la finestra applicazione gestisca operazioni su tutti i differenti tipi di oggetti?
- Ci sono oggetti «semplici» ed oggetti «compositi», ovvero contenitori di altri oggetti

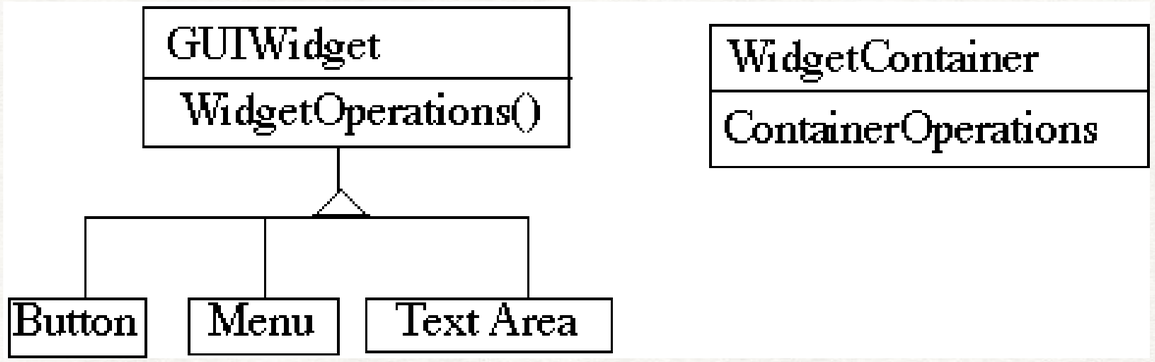
Senza pattern...

```
class Window {
    Buttons[] myButtons;
    Menus[] myMenus;
    TextAreas[] myTextAreas;
    WidgetContainer[] myContainers;

    public void update() {
        if (myButtons != null)
            for (int k = 0; k < myButtons.length(); k++) myButtons[k].refresh();
        if (myMenus != null)
            for (int k = 0; k < myMenus.length(); k++) myMenus[k].display();
        if (myTextAreas != null)
            for (int k = 0; k < myButtons.length(); k++) myTextAreas[k].refresh();
        if (myContainers != null)
            for (int k = 0; k < myContainers.length(); k++)  myContainers[k].updateElements();
        // etc.
    }

    public void fooOperation() {
        if (myButtons != null);
        // etc.
    }
    //etc...
}
```

devo ripetere più volte la stessa operazione concettuale e distinguere singolarmente tutte le diverse tipologie di oggetti cui applicarla

Un primo passo avanti

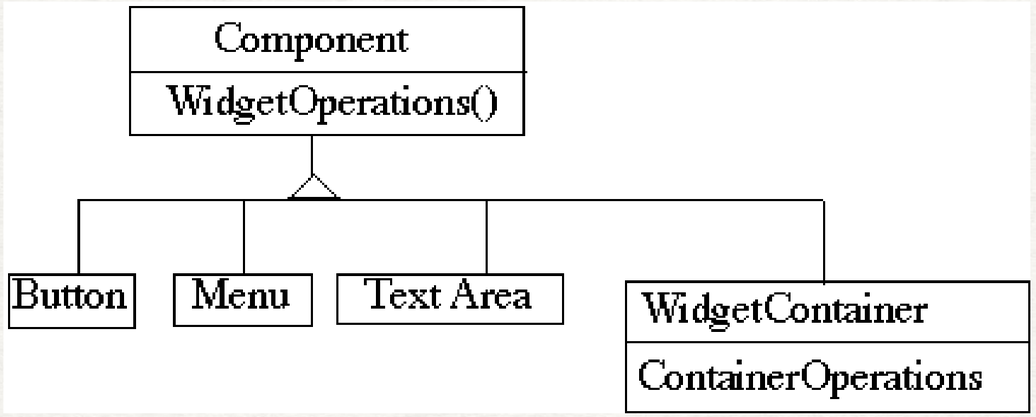
- Riduco la distinzione a solo tra oggetti semplici e contenitori
- Uso una interfaccia per astrarre i vari tipi semplici
- Continuo ad dovere accedere a tutti gli oggetti
- Non si tiene conto che sono nidificati in svariati livelli di composizione




```
class Window {
    GUIWidgets[] myWidgets;
    WidgetContainer[] myContainers;

    public void update() {
        if (myWidgets != null)
            for (int k = 0; k < myWidgets.length(); k++)
                myWidgets[k].update();
        if (myContainers != null)
            for (int k = 0; k < myContainers.length(); k++)
                myContainers[k].updateElements();
        // etc.
    }
}
```

Col pattern Composite

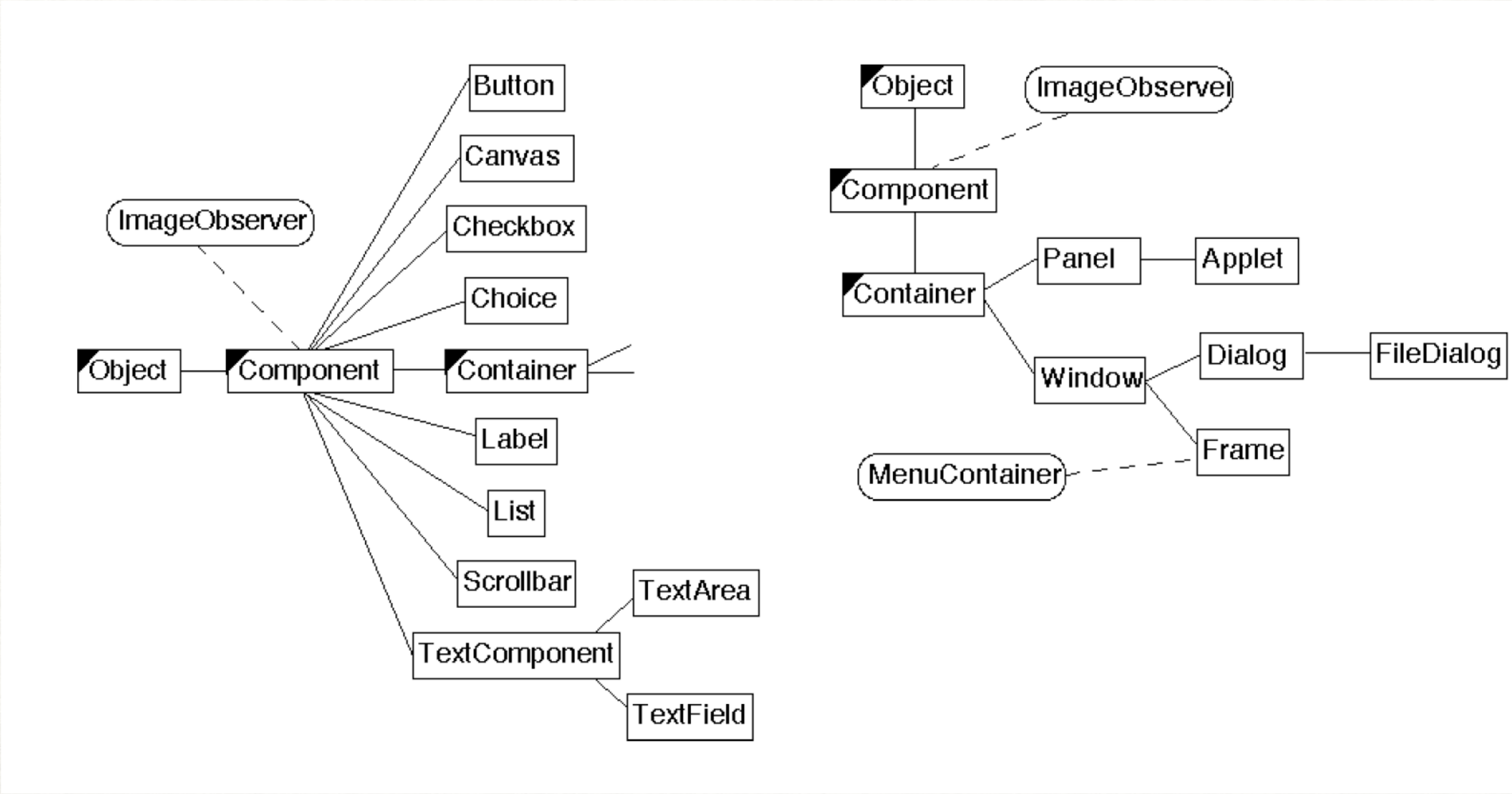
- Il Component è un'interfaccia uniforme per interagire con tutti i tipi di oggetti
- Quando possibile, implementa il comportamento di default degli oggetti semplici
- Quando necessario, gli oggetti semplici ne fanno override per specializzarli
- Il Composite (widgetContainer) è tenuto a fare override di tutti i metodi del component, per gestirne l'applicazione sugli oggetti che contiene al suo interno




```
class WidgetContainer extends Component {
    Component[] myComponents;

   //ricorsione object oriented
    public void update() {
        if (myComponents != null)
            for (int k = 0; k < myComponents.length(); k++)
                myComponents[k].update();
    }

   // add container specific operations here
}
```

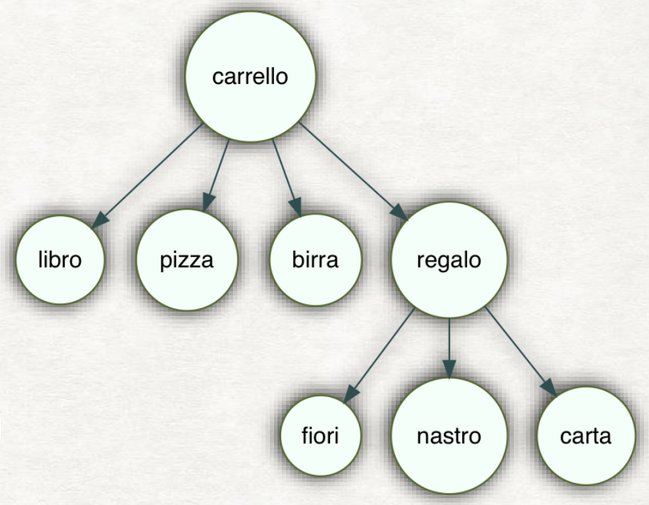
in Java

- Lungamente impiegato per le librerie AWT in Java fin dalla versione 1.0



Composite

- **Intento**: permettere la gestione ottimale di strutture ad oggetti composite, ovvero che contengono relazioni del tipo *tutto-parte* (e quindi gerarchie di oggetti, ovvero strutture ad albero).
  - Composite permette ai client di trattare oggetti singoli e composizioni di oggetti uniformemente
- **Motivazione**
  - E' necessario raggruppare elementi semplici tra loro per formare elementi più grandi
  - Se nel client devo distinguere tra classi per elementi semplici e classi per contenitori, dovendo trattarli in modo differente, questo rende il codice più complicato
  - Composite permette di descrivere una composizione ricorsiva in cui i client trattano tutti gli elementi della struttura uniformemente
- **Collaborazioni**
  - I client usano l'interfaccia di Component per interagire con elementi della struttura composita. Se il ricevente del messaggio è Leaf, la richiesta è gestita direttamente. Se il ricevente è un Composite, questo invia la richiesta ai suoi child e possibilmente avvia operazioni addizionali prima e dopo

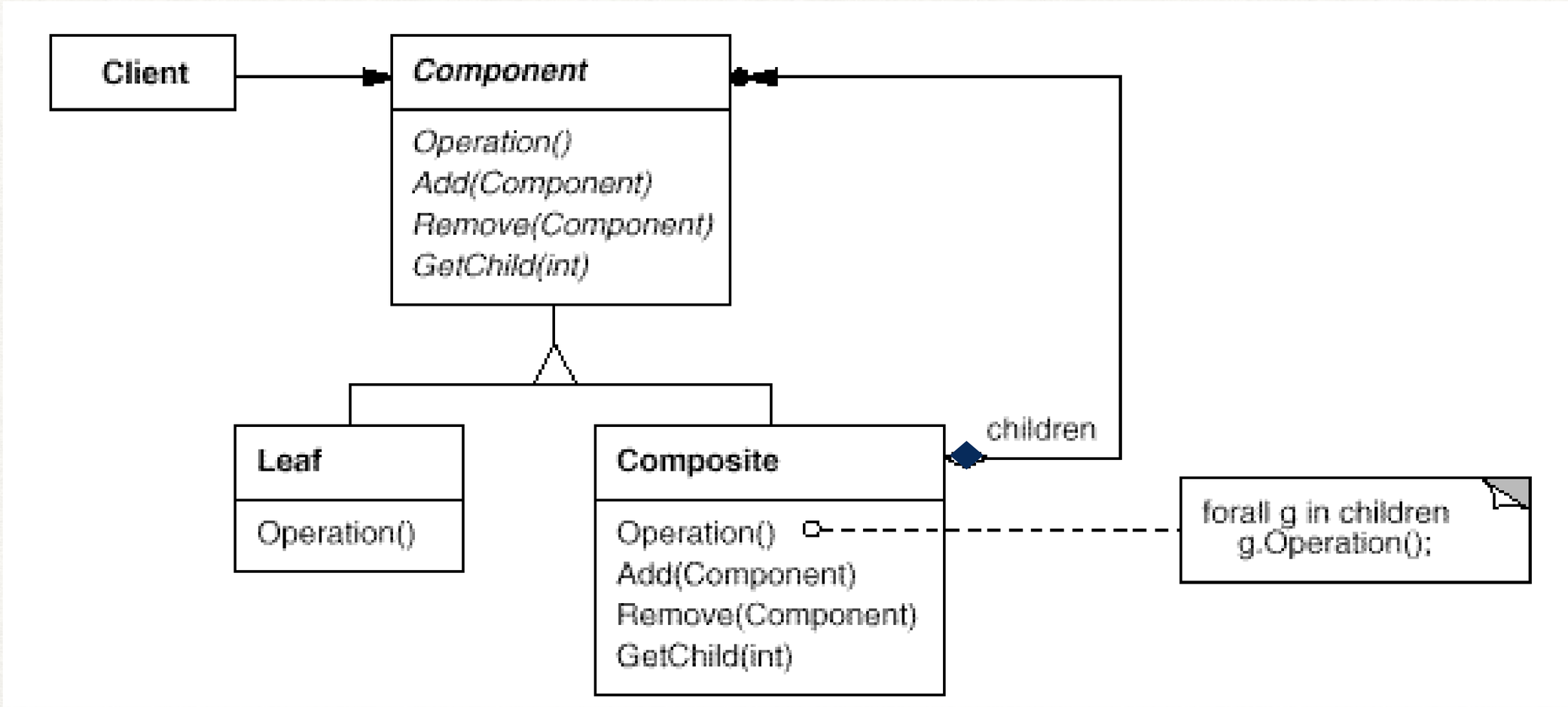


Composite

- **Soluzione**
  - **Component**: interfaccia (o classe abstract) che rappresenta uniformemente vari elementi semplici e non;
    - dichiara le operazioni comuni applicabili a tutti oggetti **semplici** della composizione;
    - può definire un'operazione che permette ad un elemento di accedere all'oggetto padre nella struttura ricorsiva
  - **Leaf**: classe che rappresenta elementi semplici; è sottoclasse di Component; implementa il comportamento degli oggetti semplici facendo override dei comportamenti di default
  - **Composite**: sottoclasse di Component che rappresenta elementi contenitori;
    - mantiene il riferimento ad un set di elementi *child (leaf o composite)*;
    - implementa operazioni per gestire il gruppo dei child (add, remove, …)
    - implementa le operazioni dichiarate in Component applicandole ricorsivamente all'aggregato dei suoi elementi child;

Composite

- Soluzione



metodi del Composite

- Il container tende ad essere solo un oggetto con funzioni di raccoglitore, non per forza anche un oggetto con caratteristiche peculiari sue
  - Le sue funzioni tipicamente sono solo correlate alla gestione dei suoi *children (add-remove)*
  - ***È giusto dichiarare tali operazioni nell'interfaccia Component?***
  - **Vantaggi**: trasparenza
    - Tutti gli oggetti hanno la stessa interfaccia e sono quindi indistinguibili per il client: lo scopo del pattern
  - **Svantaggi**: esposizione a rischio
    - add/remove fuori dal *composite* è un errore logico. Invocarle sulle leaf potrebbe anche generare un errore a run-time, altrimenti segnalato già in compilazione
    - Alternativa, implementarle sollevando una eccezione o controllando a run-time il tipo dell'oggetto prima di invocarle (perdita di trasparenza)

Composite

- Si può inserire una operazione *getComposite*() in Component che ritorna null e che è ridefinita in Composite per ritornare il riferimento a se stesso. I client dovrebbero comunque distinguere il tipo di risultato e fare operazioni differenti, niente trasparenza
- La lista che contiene elementi child deve essere definita in Composite, altrimenti se fosse definita in Component si sprecherebbe spazio, poiché ogni Leaf avrebbe tale variabile anche se non deve usarla mai
- L'ordinamento di child per un Composite potrebbe essere importante e va tenuto in considerazione su certe implementazioni
- Il Composite potrebbe implementare una cache, per ottimizzare le prestazioni, è utile quando si implementa un'operazione che deve ricercare tra tutti i suoi componenti child. I componenti child devono poter accedere ad un'operazione che permette di invalidare la cache del Composite

Riferimenti espliciti al padre

- Per facilitare la navigazione tra gli oggetti, gli elementi child mantengono il riferimento all'oggetto che li contiene
- Il riferimento è inserito in Component, mentre Leaf e Composite possono implementare le operazioni per gestirlo

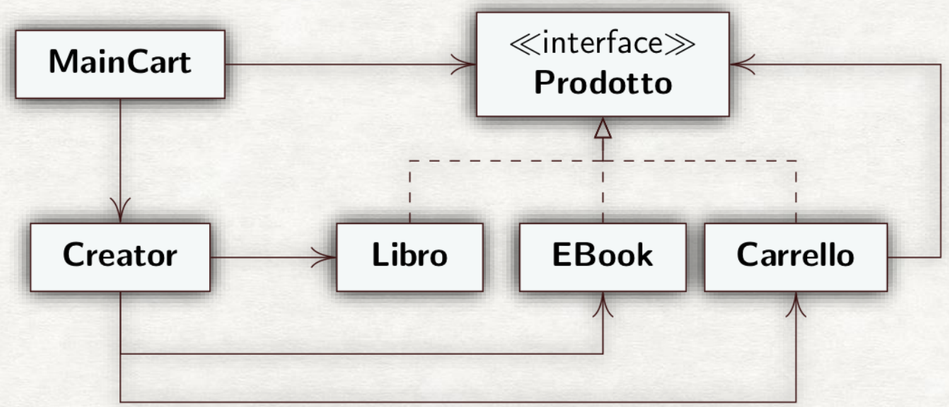
```
class WidgetContainer extends Component {
    Component[] myComponents;

    public void update() {
        if (myComponents != null)
            for (int k = 0; k < myComponents.length(); k++)
                myComponents[k].update();
    }
    public void add(Component aComponent) {
        myComponents.append(aComponent);
        aComponent.setParent(this);
    }
}

abstract class Component {
    protected Component parent;
    public void setParent(Component myParent) {
        parent = myParent;
    }
    // etc.
}
```

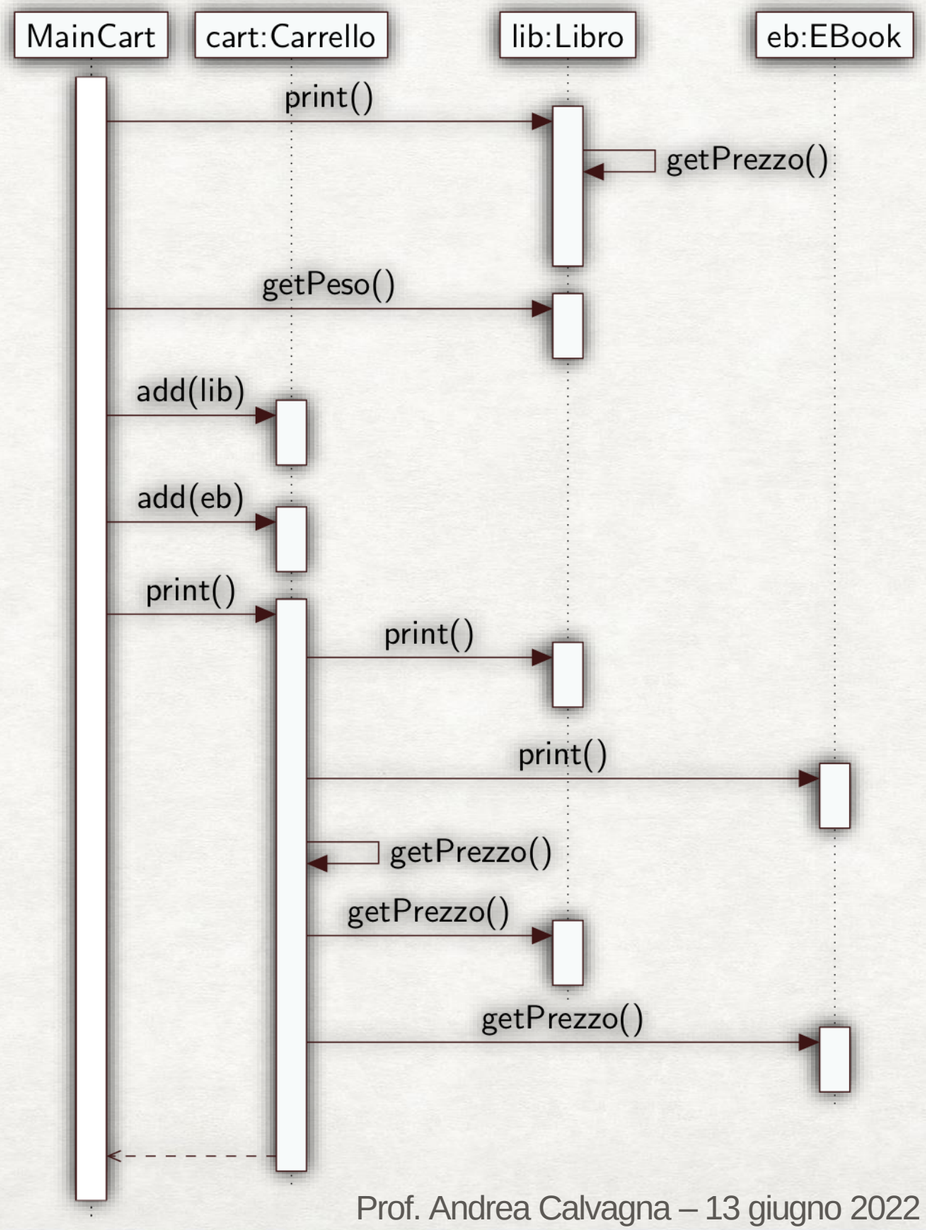
Conseguenze

- Elementi semplici possono essere composti in elementi più complessi, questi possono essere composti, e così via (ovvero composizione ricorsiva)
- **Un client che si aspetta un elemento semplice può riceverne uno composto**
- I client sono semplici, trattano strutture composte e semplici uniformemente. I client non devono sapere se trattano con un Leaf o un Composite
- **Nuovi tipi di elementi (Leaf o Composite) possono essere aggiunti e potranno funzionare con la struttura ed i client esistenti**
- Non è possibile a design time vincolare il Composite solo su certi componenti Leaf (in base al tipo), invece dovranno essere fatti dei controlli a runtime

Esempio

- Si vuol gestire un prodotto e un insieme di prodotti (nel carrello) allo stesso modo





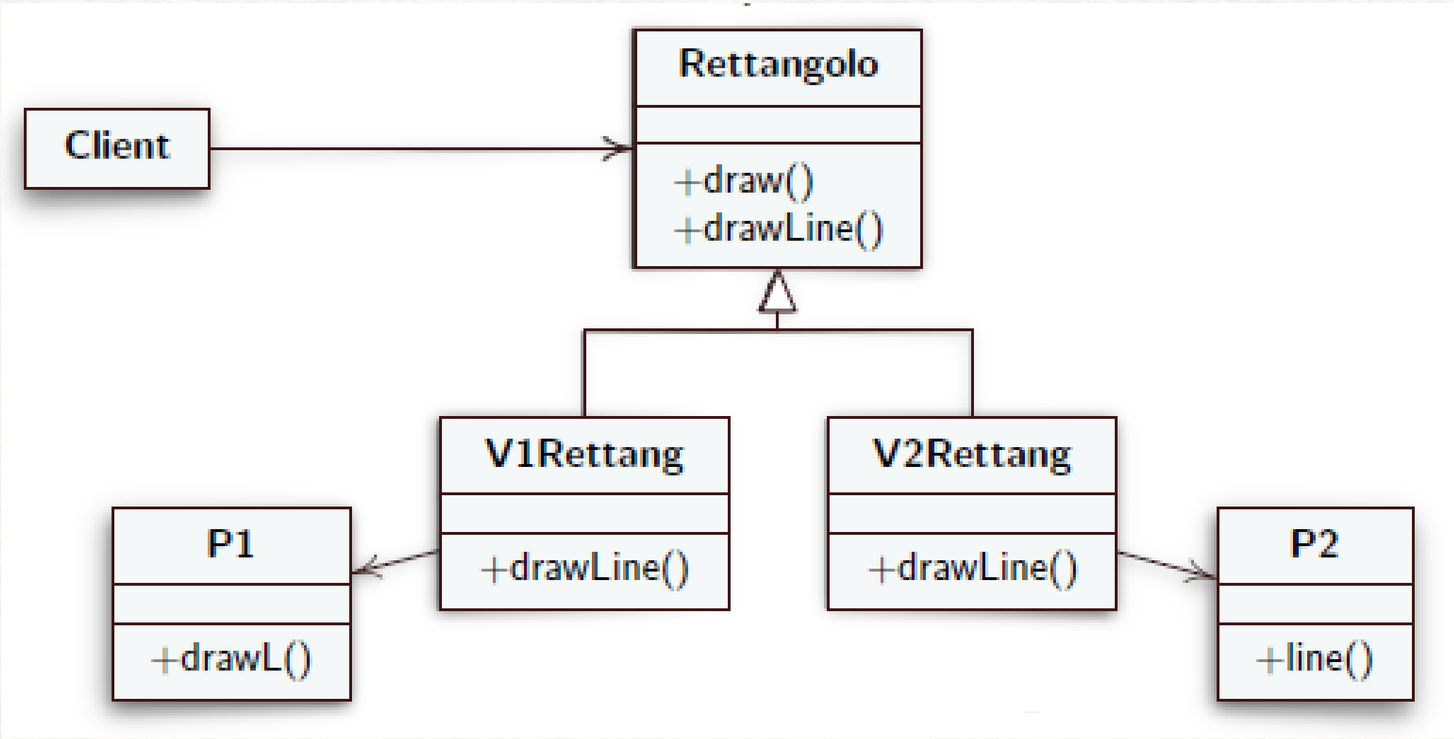
Pausa 10 min

BRIDGE

- Intento
  - Disaccoppiare una astrazione dalla sua implementazione così che le due possano variare indipendentemente
- Motivazione
  - Quando una astrazione può avere varie implementazioni, di solito si usa l'ereditarietà. Una classe astratta definisce l'interfaccia dell'astrazione, le sottoclassi concrete la implementano in modi diversi
  - Questo approccio non è flessibile poiché collega l'astrazione all'implementazione permanentemente, e rende difficile modificare, estendere e usare astrazioni e implementazioni in modo indipendente le une dalle altre

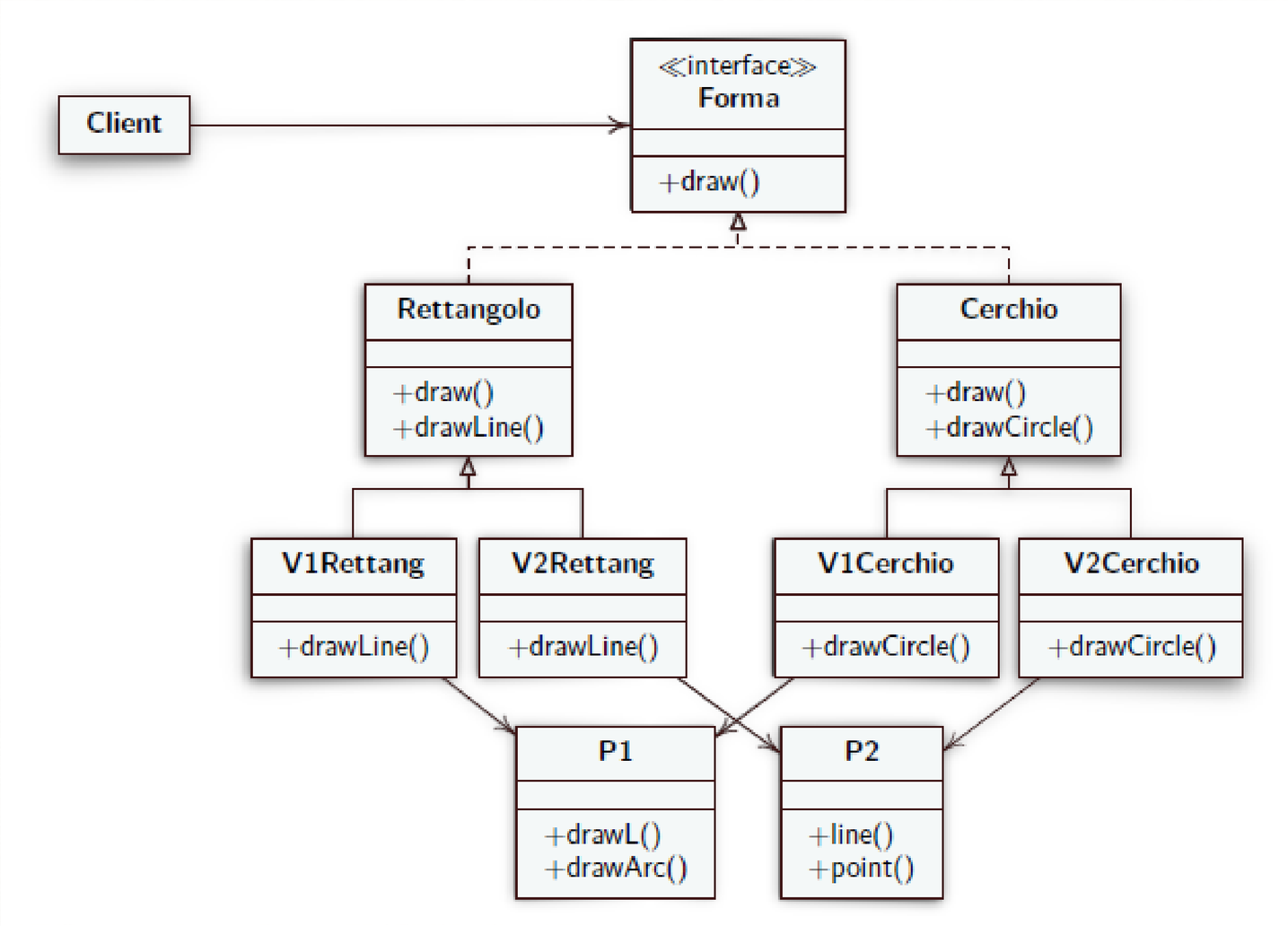
Esempio

- In un sistema, occorre avere Rettangoli e Cerchi (astrazioni).
- Inoltre, occorre che Rettangoli e Cerchi siano disponibili per due piattaforme P1 e P2 (implementazioni)
- La piattaforma P1 mette a disposizione drawL() per disegnare linee, mentre la piattaforma P2 fornisce line() •
- Struttura classi iniziale (prima di usare Bridge)



ESEMPIO

- Struttura classi iniziale (prima di usare Bridge) con l'aggiunta del cerchio e l'uso dell'astrazione forma

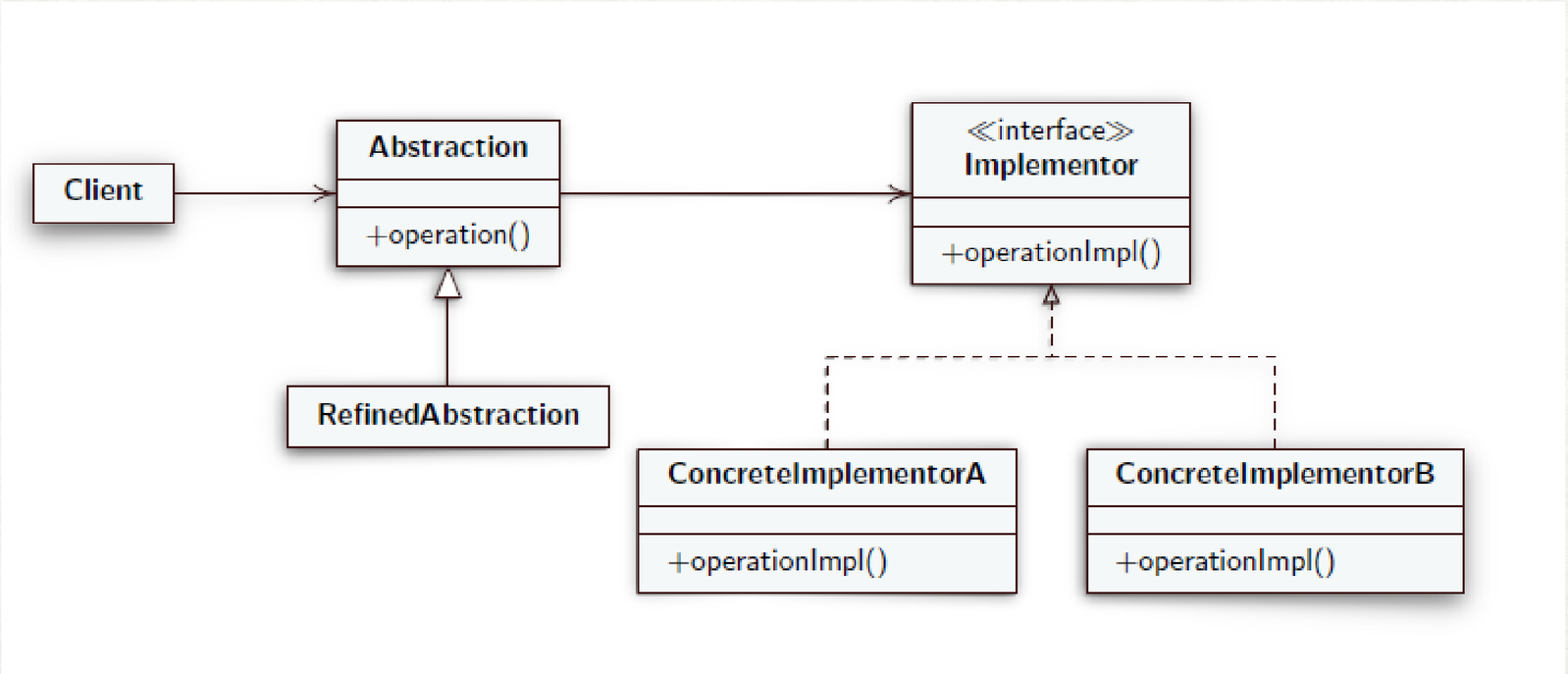


Considerazioni

- La soluzione appena vista porterebbe ad una proliferazione di classi
- Se introducessi un'altra piattaforma P3 (implementazione) avrei bisogno di 6 classi concrete (2 forme x 3 piattaforme)
- Se introducessi un'altra forma (astrazione), avrei bisogno di 9 classi concrete (3 x 3)
- Per ogni variazione da introdurre vorrei invece un incremento lineare del numero di classi
- Inoltre, presa una classe, per esempio la classe V1Rettang, essa è legata a una certa piattaforma, es. P1, in modo permanente, ovvero una istanza di V1Rettang non può usare una piattaforma diversa da P1

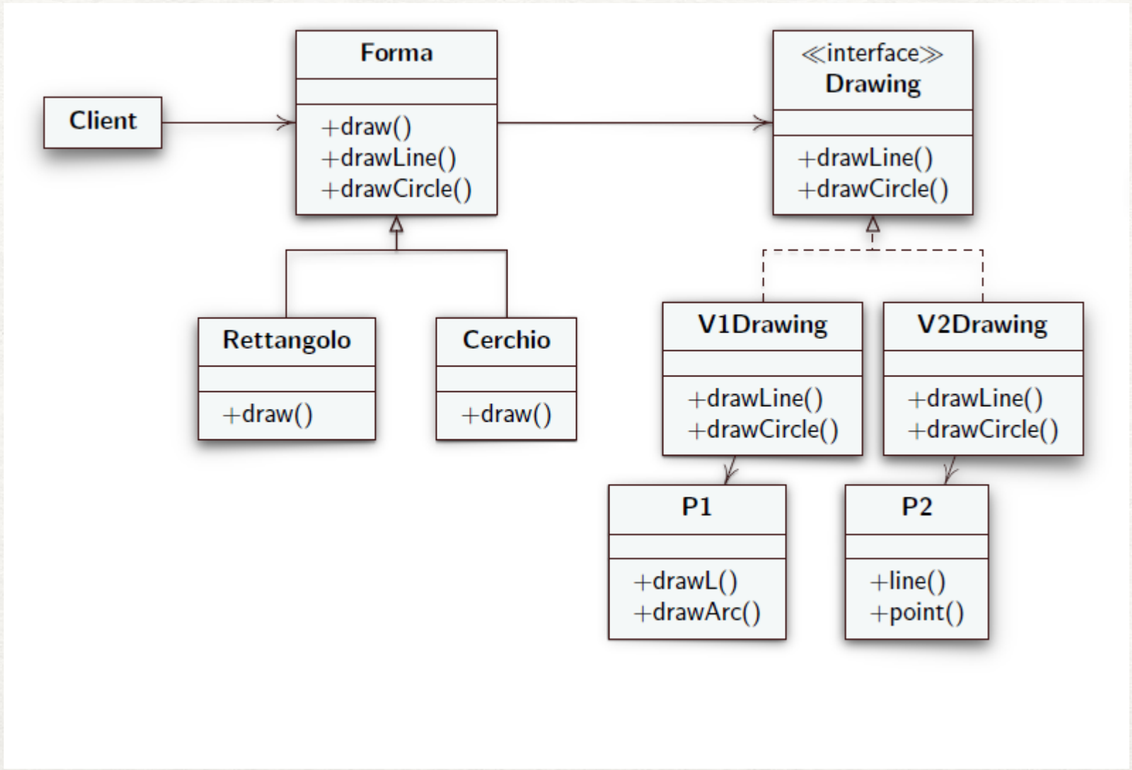
Bridge

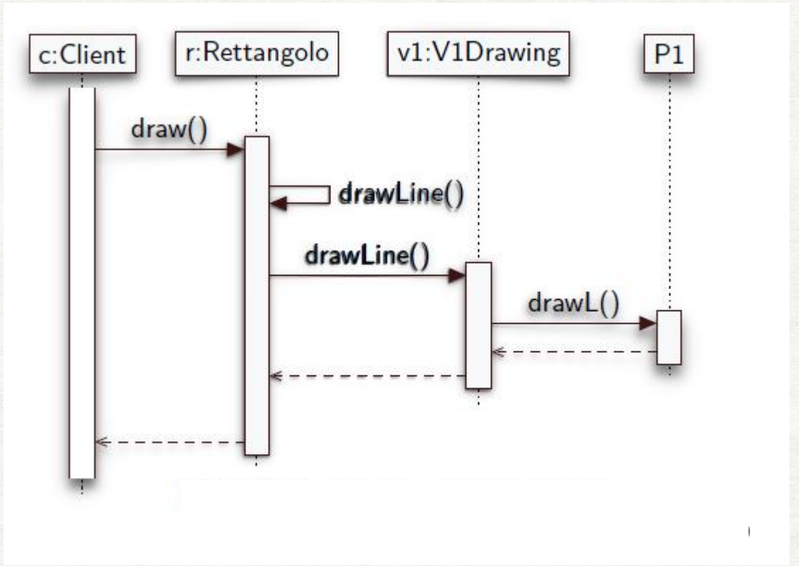
- **Abstraction** definisce l'interfaccia per i client e mantiene un riferimento ad un oggetto di tipo **Implementor**. **Abstraction** inoltra le richieste del client al suo oggetto **Implementor**
- **RefinedAbstraction** estende l'interfaccia definita da **Abstraction**
- **Implementor** definisce l'interfaccia per le classi dell'implementazione. Questa interfaccia non deve corrispondere esattamente ad **Abstraction**, di solito **Implementor** fornisce operazioni primitive, mentre **Abstraction** definisce operazioni di più alto livello basate su tali primitive
- **ConcreteImplementor** implementa l'interfaccia di Implementor e fornisce le operazioni concrete



Bridge

- La soluzione suggerita dal design pattern Bridge, per il precedente esempio, avrà il diagramma delle classi disegnato sotto
  - Rettangolo.draw() chiama drawLine() della superclasse Forma, quest'ultima chiama drawLine() su un'istanza di una sottoclasse di Drawing. Analoghe chiamate avvengono per Cerchio
  - Una nuova classe, es. Rombo, sarà implementata come sottoclasse di Forma. La classe Rombo chiamerà drawLine() su Forma. Non occorre nessuna nuova classe della gerarchia Implementor





Conseguenze

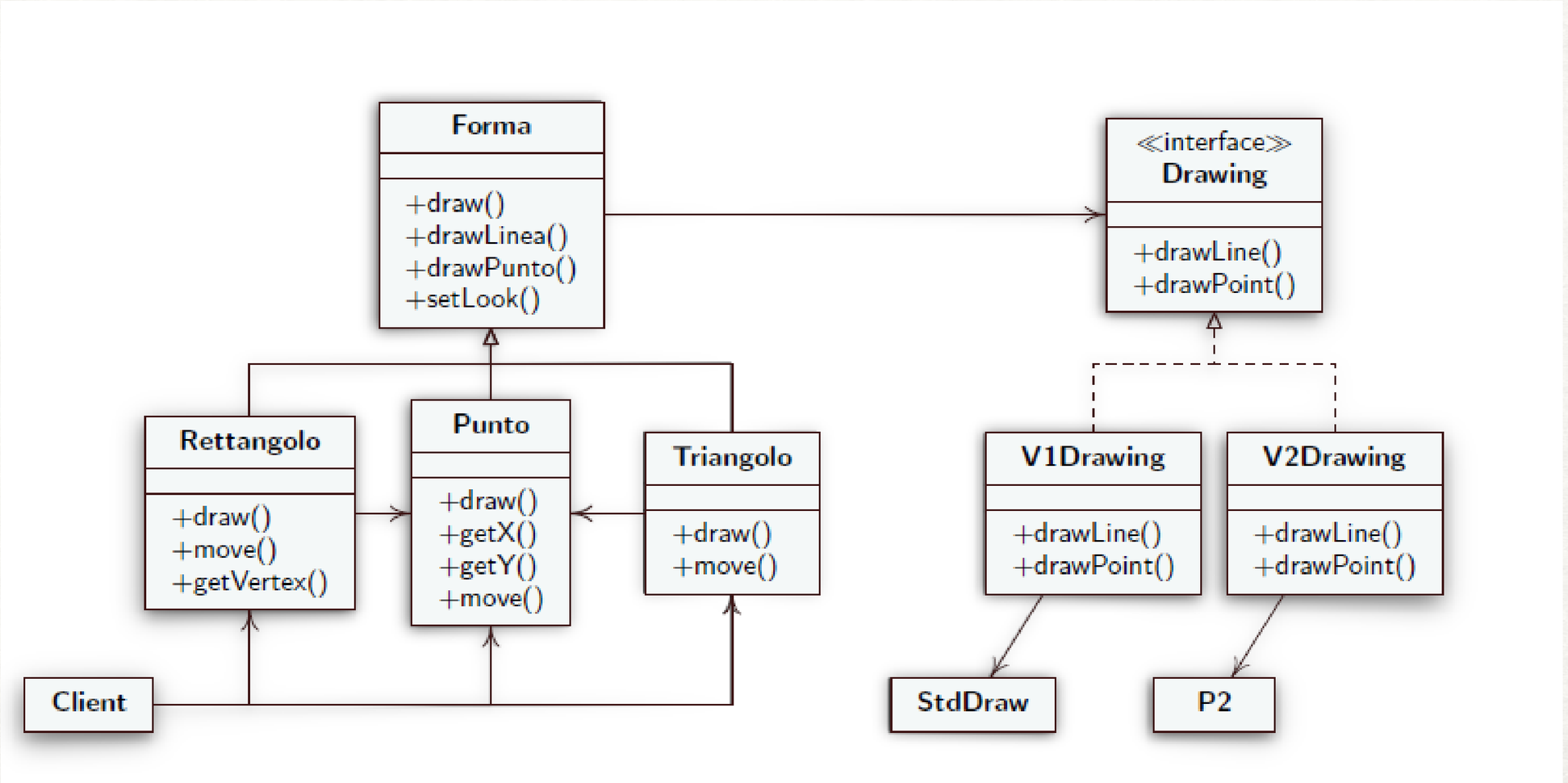
- Bridge permette a una implementazione di non essere connessa permanentemente a una interfaccia, l'implementazione può essere configurata e anche cambiata a runtime
- Il disaccoppiamento permette di cambiare l'implementazione senza dover ricompilare Abstraction ed i Client
- Solo certi strati del software devono conoscere Abstraction e Implementor
- I Client non devono conoscere Implementor
- Le gerarchie di Abstraction e Implementor possono evolvere in modo indipendente

Esempio minimale

```
// Forma è una Abstraction
public class Forma {
      private Drawing impl;
      public void setImplementor(Drawing imp) { this.impl = imp; }
      public void drawLine(int x, int y, int z, int t) { impl.drawL(x, y, z, t); }
}

// Drawing è un Implementor public interface Drawing {
      public void drawL(int x1, int y1, int x2, int y2);
}

// Rettangolo è una RefinedAbstraction
public class Rettangolo extends Forma {
      private int a, b, c, d;
      public Rettangolo(int xi, int yi, int xf, int yf) { a = xi; b = yi; c = xf; d = yf; }
      public void draw() {
            drawLine(a, b, c, b);
            drawLine(a, b, a, d);
            drawLine(c, b, c, d);
            drawLine(a, d, c, d);
      }
}
```

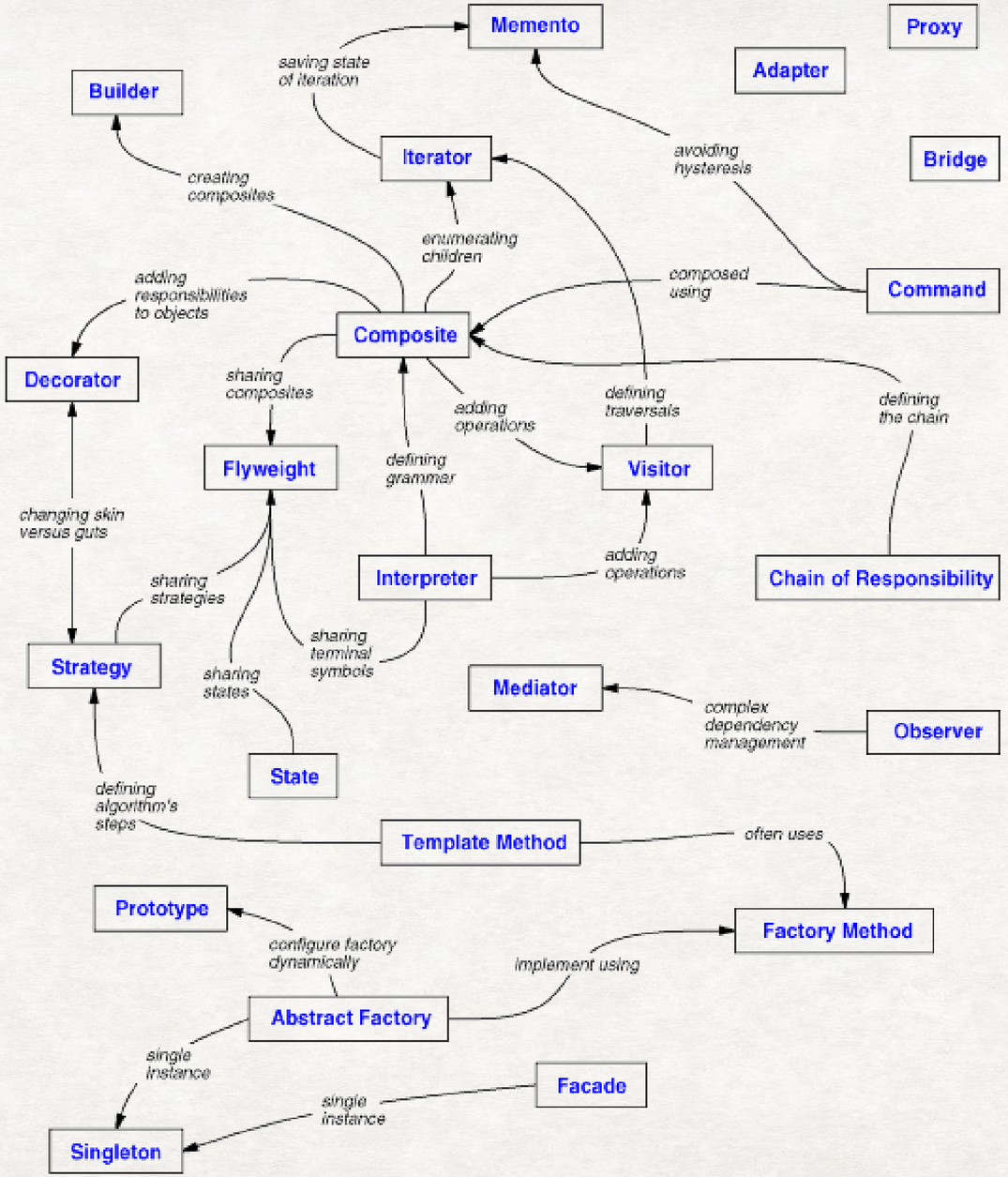
Versione più completa

- Estendo l'astrazione, aggiungendo il punto e il triangolo, senza impatto sull'implementazione
- Cambio la piattaforma cui fa riferimento l'impl.V1Drawing, senza impatto sull'astrazione



Bridge vs Adapter

- Situazioni molto diverse:
  - Bridge è una scelta che va applicata fin dall'inizio, a design time, per avere la flessibilità di poter combinare ed evolvere separatamente implementazione ed astrazione
  - Adapter affronta il problema inverso: voglio coniugare **una** astrazione ed **una** implementazione, senza alterarle, poichè sono state già progettate e realizzate, separatamente ma senza alcun legame tra loro.
  - Adapter è anche una soluzione per far funzionare insieme classi completamente scorrelate tra loro
  - Con l'adapter posso anche collegare l'astrazione con più alternative di implementazione, ma devo cambiare l'adapter
  - Il bridge permette di poter estendere l'astrazione a piacimento successivamente all'introduzione del pattern, mentre l'adapter assume l'astrazione come punto fermo.


Memento
Proxy
saving state of iteration
Builder
Adapter
Iterator
avoiding hysteresis
Bridge
creating composites
enumerating children
adding responsibilities to objects
composed using
Command
Composite
Decorator
sharing composites
adding operations
defining traversals
defining the chain
Flyweight
defining grammar
Visitor
changing skin versus guts
adding operations
sharing strategies
Interpreter
Chain of Responsibility
sharing terminal symbols
Strategy
sharing states
Mediator
complex dependency management
Observer
State
defining algorithm's steps
Template Method
often uses
Prototype
Factory Method
configure factory dynamically
implement using
Abstract Factory
single instance
Facade
single instance
Singleton