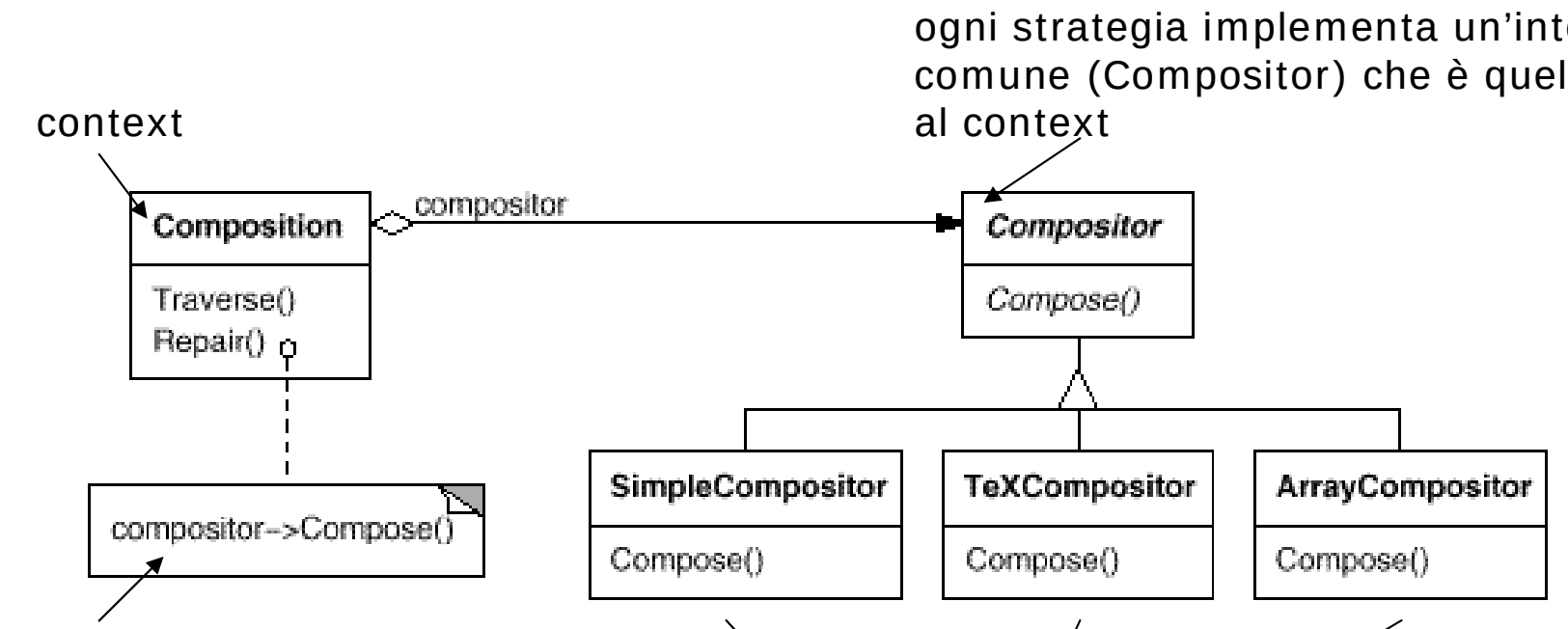
Strategy – Esempio

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

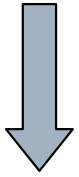


Strategy – Soluzione (ii)

```
void Composition::Repair () {
    switch (_breakingStrategy) {
        case SimpleStrategy:
                ComposeWithSimpleCompositor();
                break;
        case TeXStrategy:
                ComposeWithTeXCompositor();
                break;
        // ...
    }
}
```

aggiungere o rimuovere una strategy richiede di modificare la funzione Repair() della classe Composition

questo è un esempio semplice, in generale una lunga sequenza di istruzioni di controllo è a rischio di errori



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

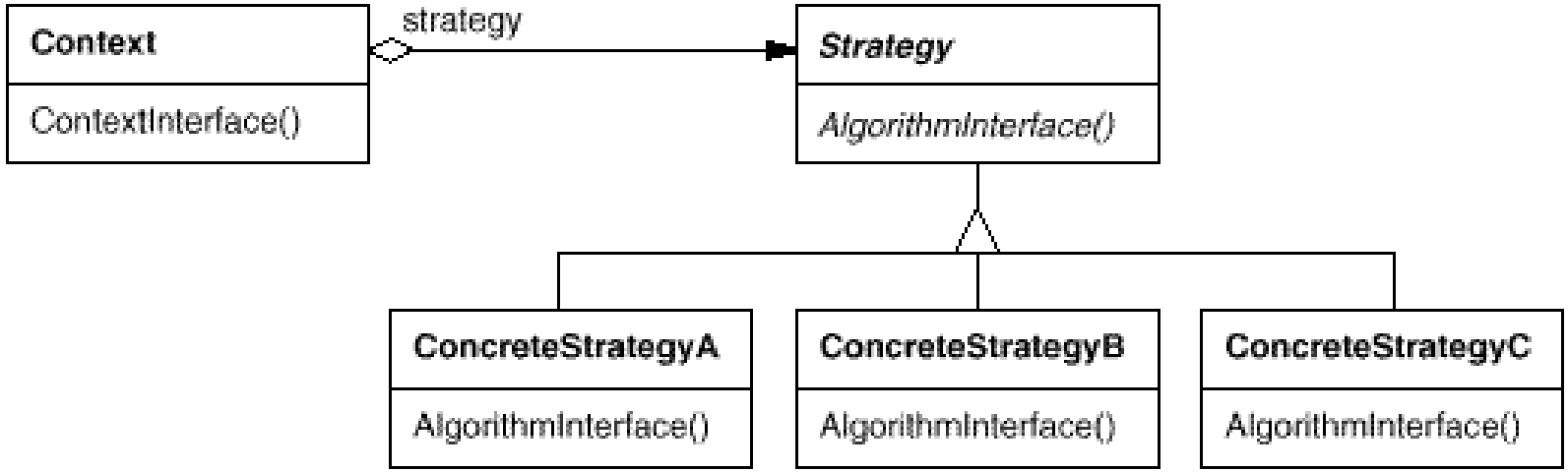
Strategy – Soluzione (iii)

```
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



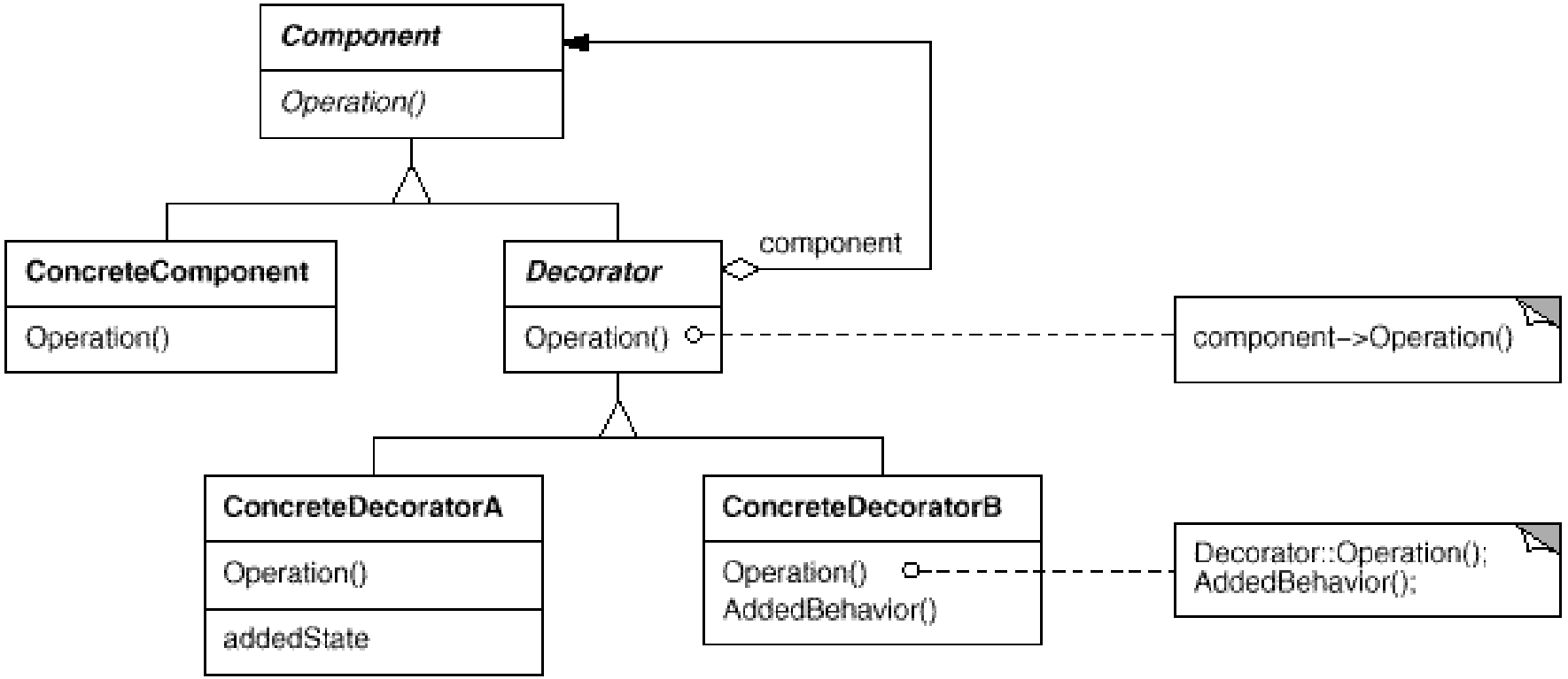
Decorator (object / structural)

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

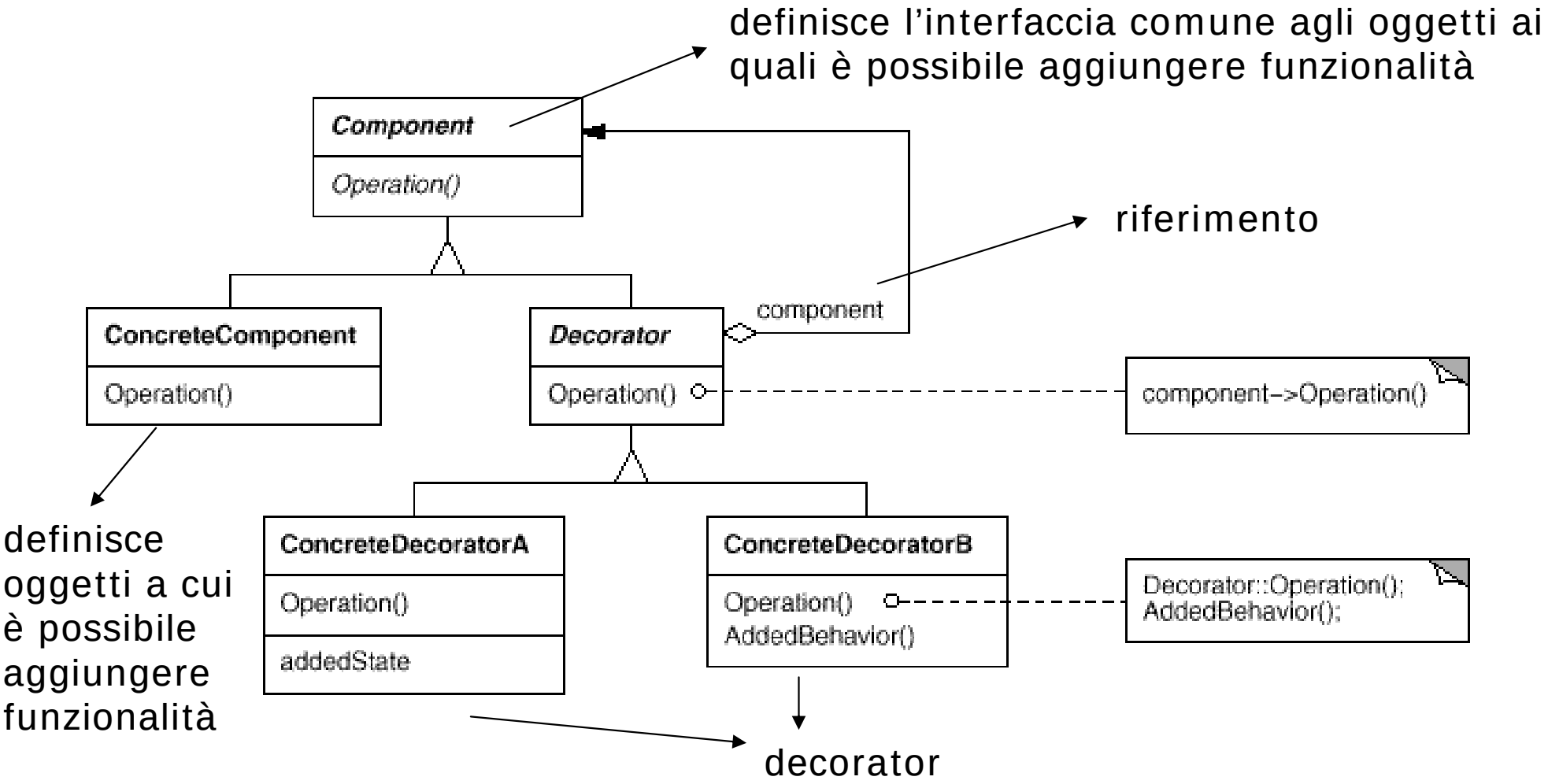
Come si possono estendere le funzionalità di un oggetto?

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)



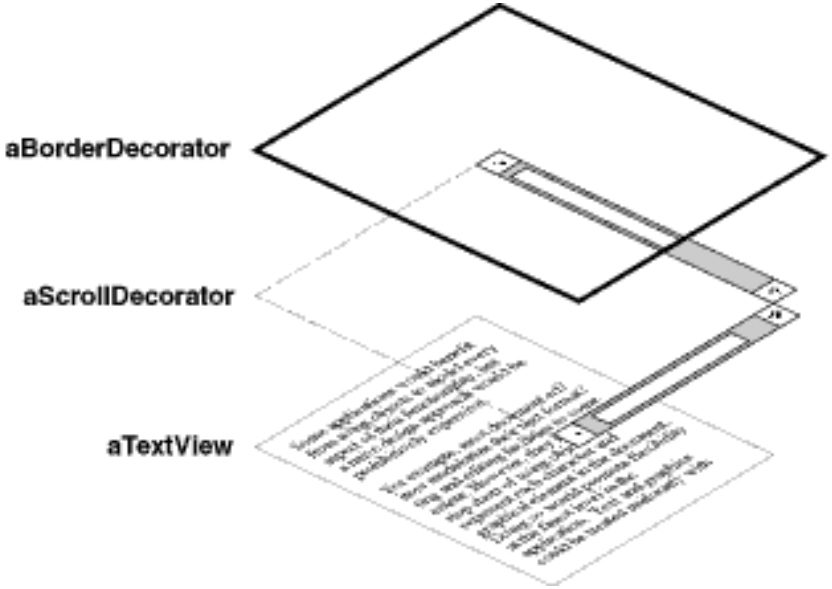
Decorator (iii)

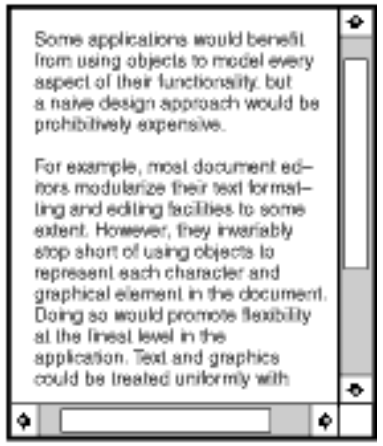


Decorator – Esempio

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();
          // ...
};
```

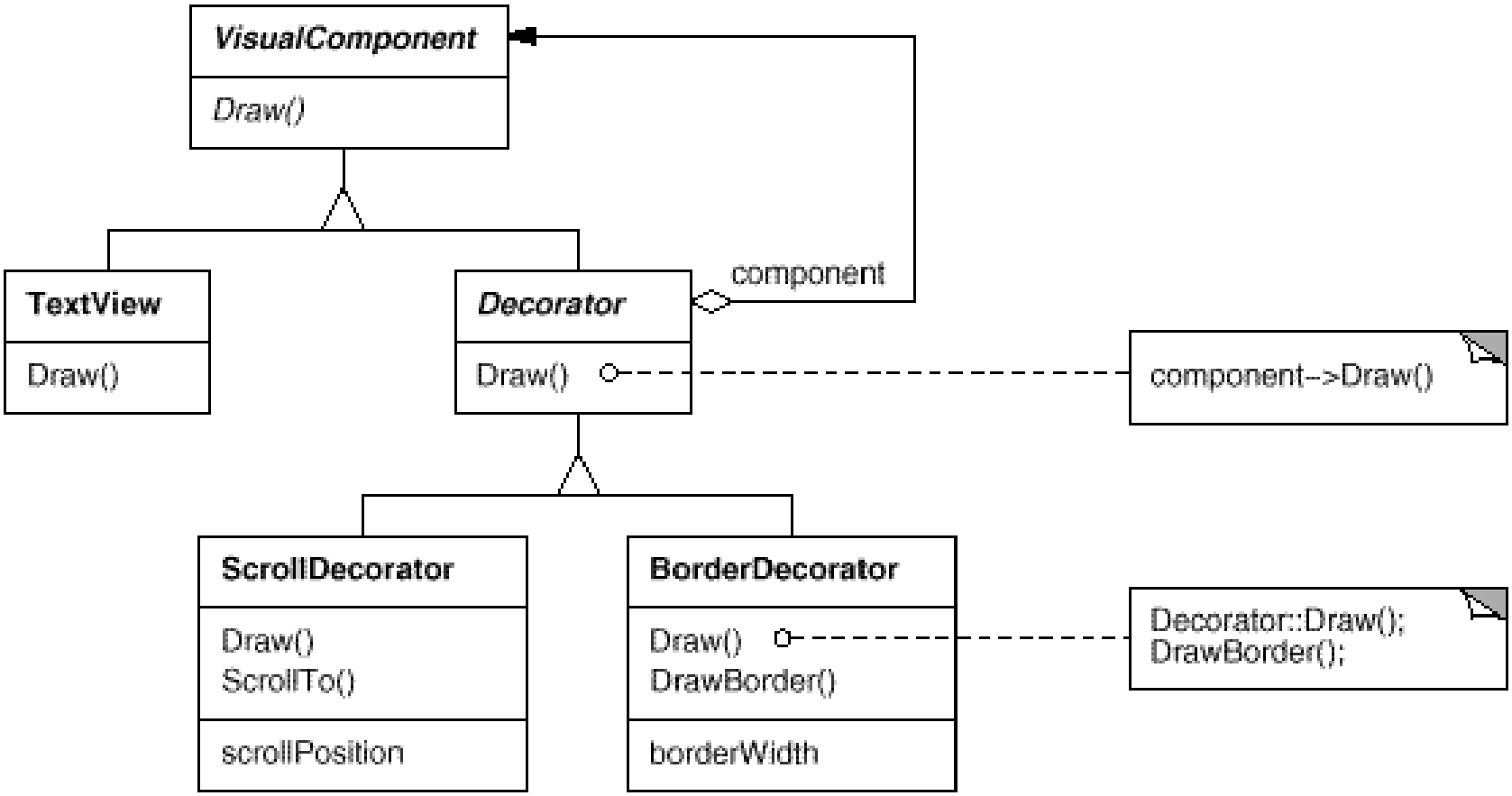
```
class TextView:public VisualComponent {
   public:
          TextView();
          virtual void Draw();
          // ...
};
```

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);
}
```