

# **Laboratorio di Progettazione di Sistemi Software**

## **Design Patterns**

### **Strutturali**

**Valentina Presutti (A-L)**

**Riccardo Solmi (M-Z)**

# Indice degli argomenti

---

- **Catalogo di Design Patterns strutturali:**
  - Composite
  - Decorator
  - Adapter

# Composite

---

- **Intent**

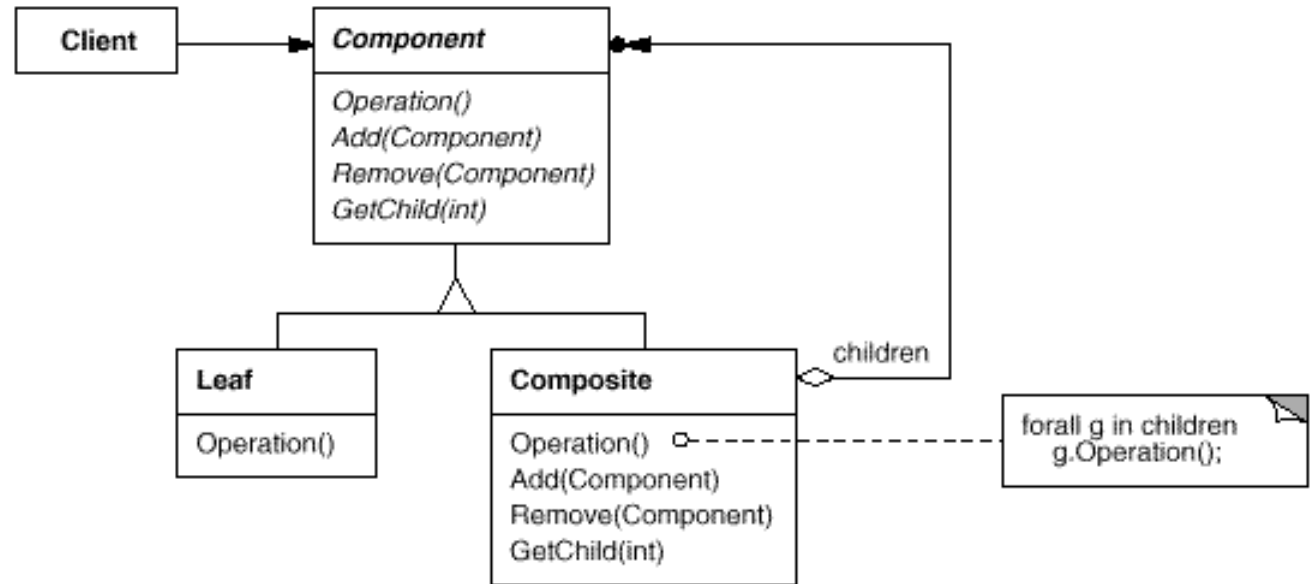
- Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly

- **Applicability**

- you want to represent part-whole hierarchies of objects.
- you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly

# Composite /2

## ■ Structure



## ■ Participants

### ■ Component

- declares the interface for objects in the composition.
- implements default behavior for the interface common to all classes.
- declares an interface for accessing and managing its child components.
- (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.

### • Leaf

- represents leaf objects in the composition.
- defines behavior for primitive objects

### • Composite

- defines behavior for components having children.
- stores child components.
- Implements child-related operations in the Component interface.

### • Client

- manipulates objects in the composition through the Component interface.

# Composite /3

---

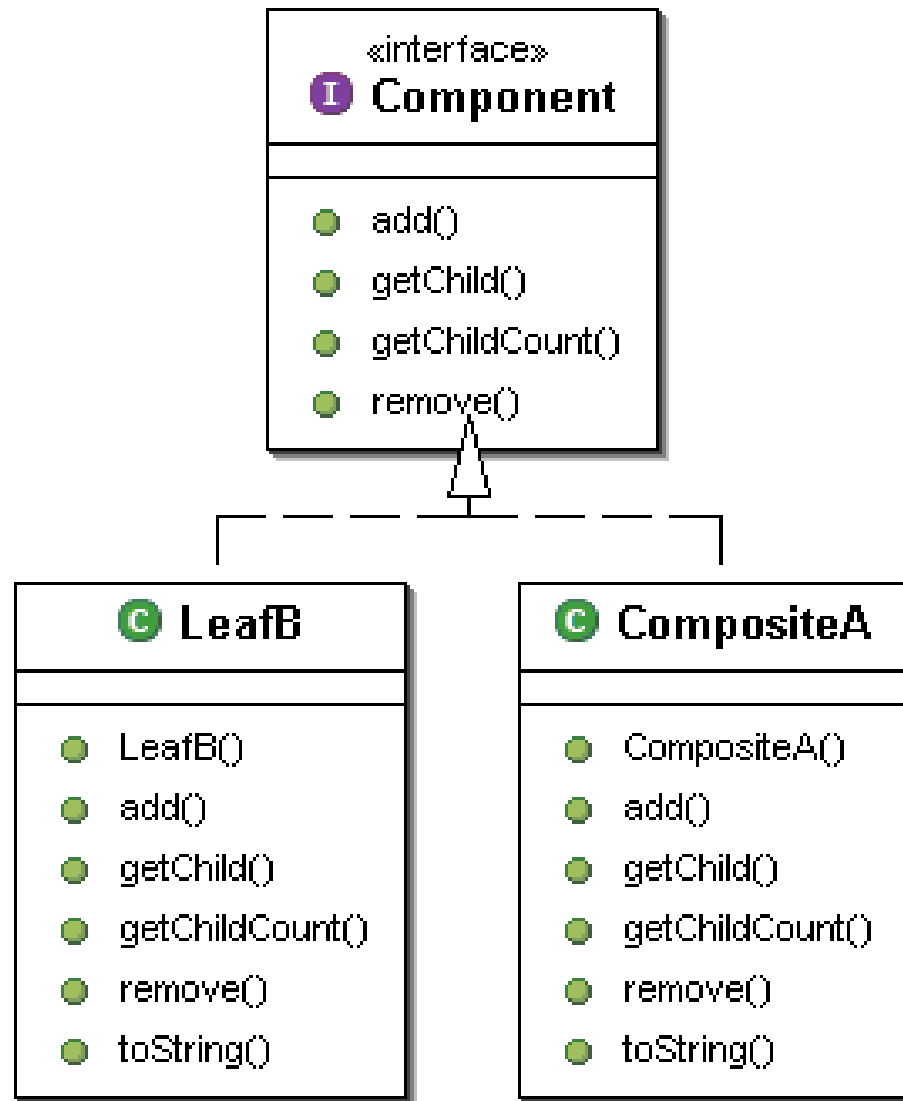
## ■ Collaborations

- Clients use the Component class interface to interact with objects in the composite structure. If the recipient is a Leaf, then the request is handled directly. If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.

## ■ Consequences

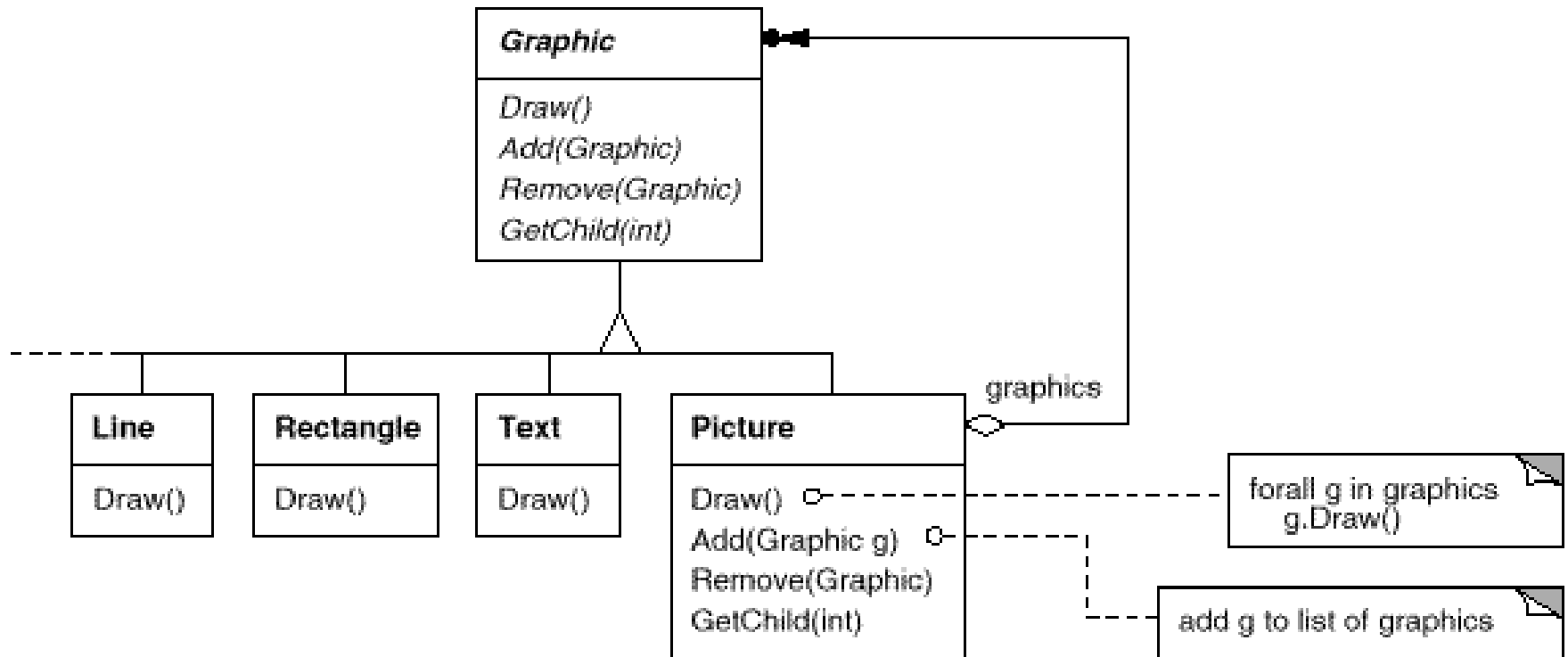
- defines class hierarchies consisting of primitive objects and composite objects
- makes the client simple
- makes it easier to add new kinds of components
- can make your design overly general

# Composite example 1



# Composite example 2

- Graphics applications like drawing editors and schematic capture systems let users build complex diagrams out of simple components



# Composite Questions

---

- **Part 1: How does the Composite pattern help to consolidate system-wide conditional logic?**
- **Part 2: Would you use the composite pattern if you did not have a part-whole hierarchy? In other words, if only a few objects have children and almost everything else in your collection is a leaf (a leaf can have no children), would you still use the composite pattern to model these objects?**



# Decorator or Wrapper

---

## ■ Intent

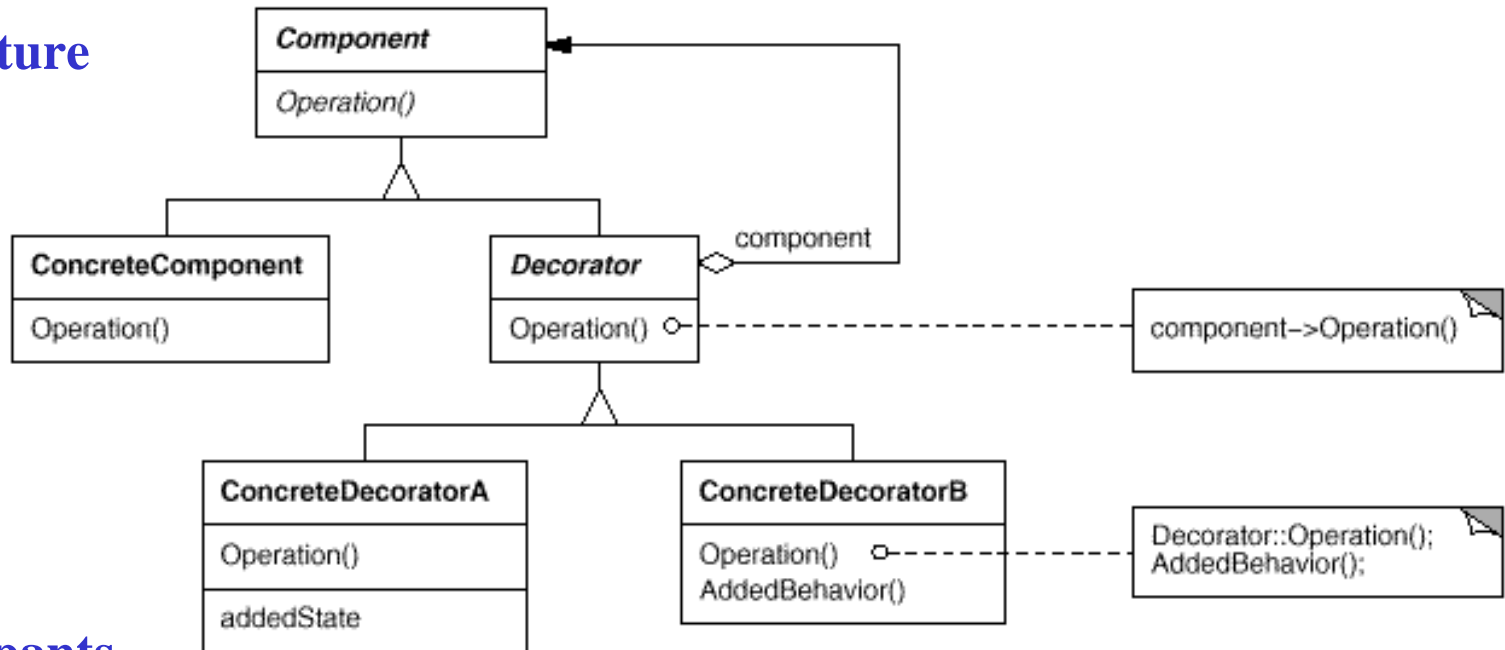
- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

## ■ Applicability

- to add responsibilities to *individual* objects *dynamically* and *transparently*, that is, without affecting other objects.
- for responsibilities that can be withdrawn.
- when extension by subclassing is impractical or not allowed

# Decorator /2

## Structure



## Participants

### • Component

- defines the interface for objects that can have responsibilities added to them dynamically

### • ConcreteComponent

- defines an object to which additional responsibilities can be attached

### • Decorator

- maintains a reference to a Component object and defines an interface that conforms to Component's interface

### • ConcreteDecorator

- adds responsibilities to the component

# Decorator /3

---

## ■ Collaborations

- Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request

## ■ Consequences

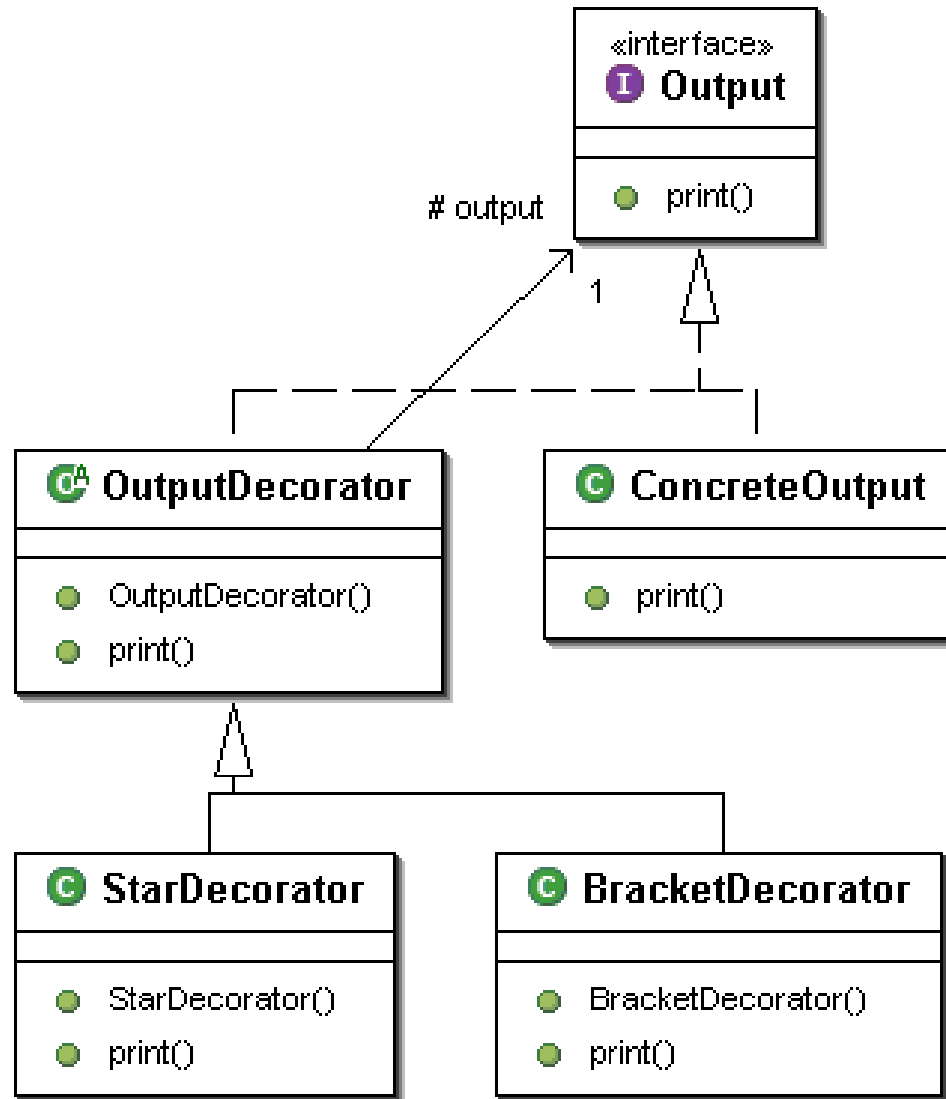
- More flexibility than static inheritance.
- Avoids feature-laden classes high up in the hierarchy.
- A decorator and its component aren't identical.
- Lots of little objects.

## ■ Implementation

- Keeping Component classes lightweight.
- Changing the skin of an object versus changing its guts.

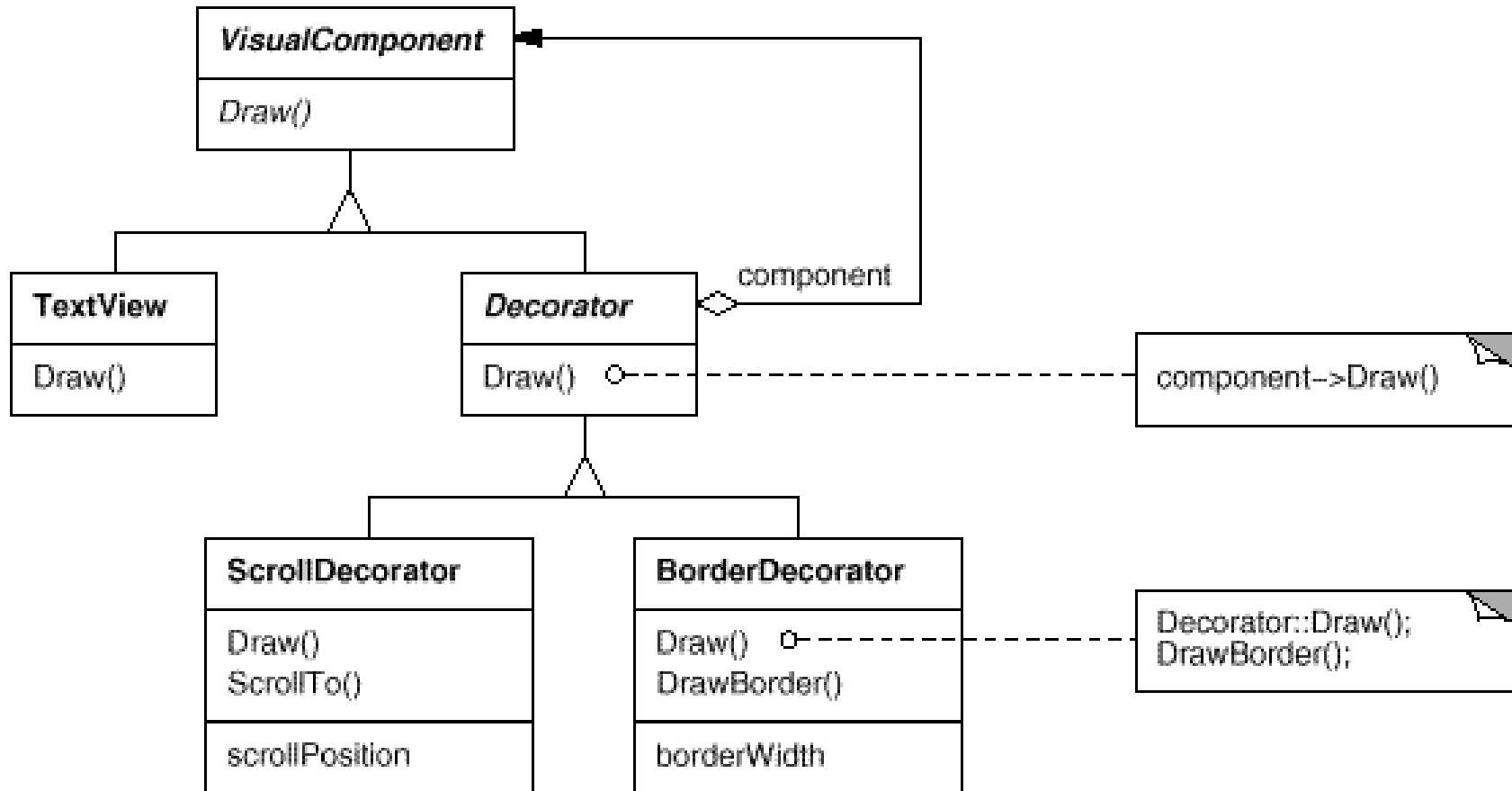
# Decorator example 1

- String Decorator



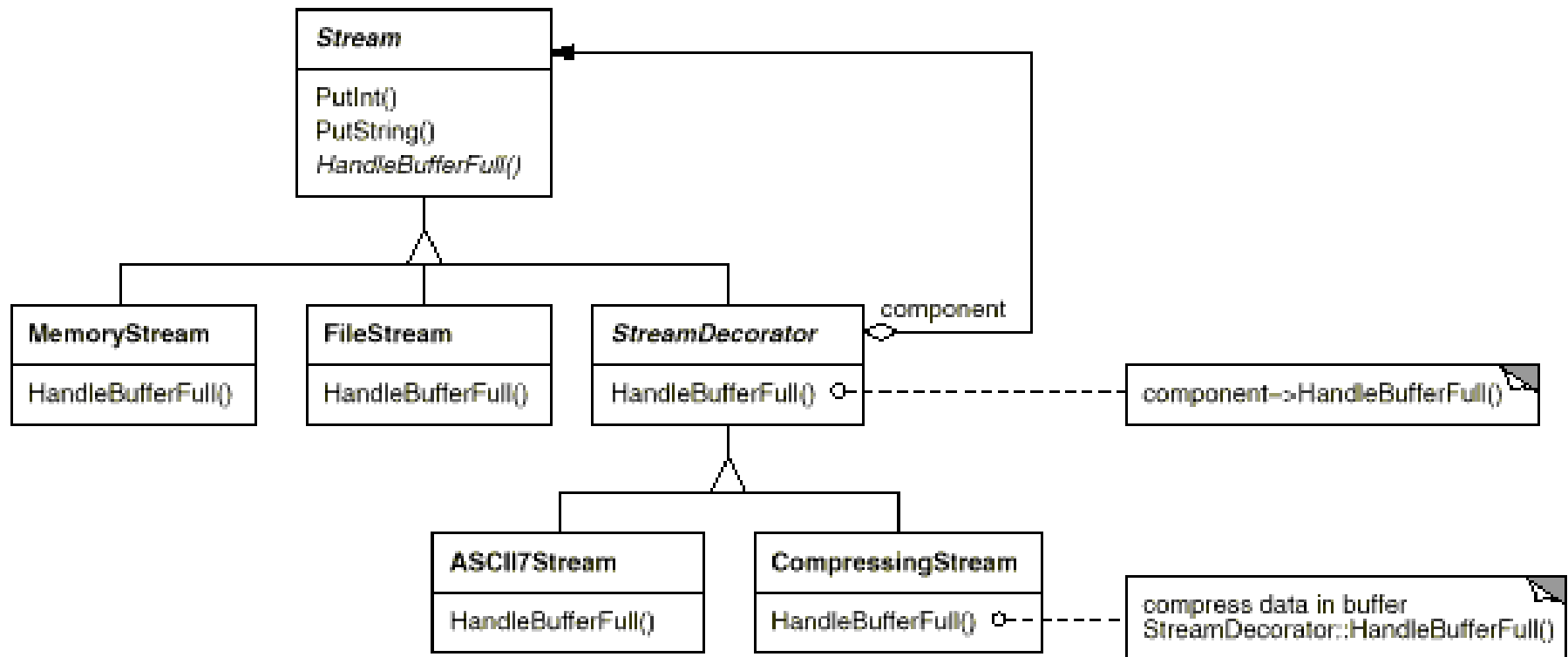
# Decorator example 2

- To decorate an individual objects



# Decorator example 3

- Adding responsibilities to streams



# Decorator questions

---

- **Now consider an object A, that is decorated with an object B. Since object B "decorates" object A, object B shares an interface with object A. If some client is then passed an instance of this decorated object, and that method attempts to call a method in B that is not part of A's interface, does this mean that the object is no longer a Decorator, in the strict sense of the pattern? Furthermore, why is it important that a decorator object's interface conforms to the interface of the component it decorates?**

# Adapter

---

## ■ Intent

- Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

## ■ Applicability

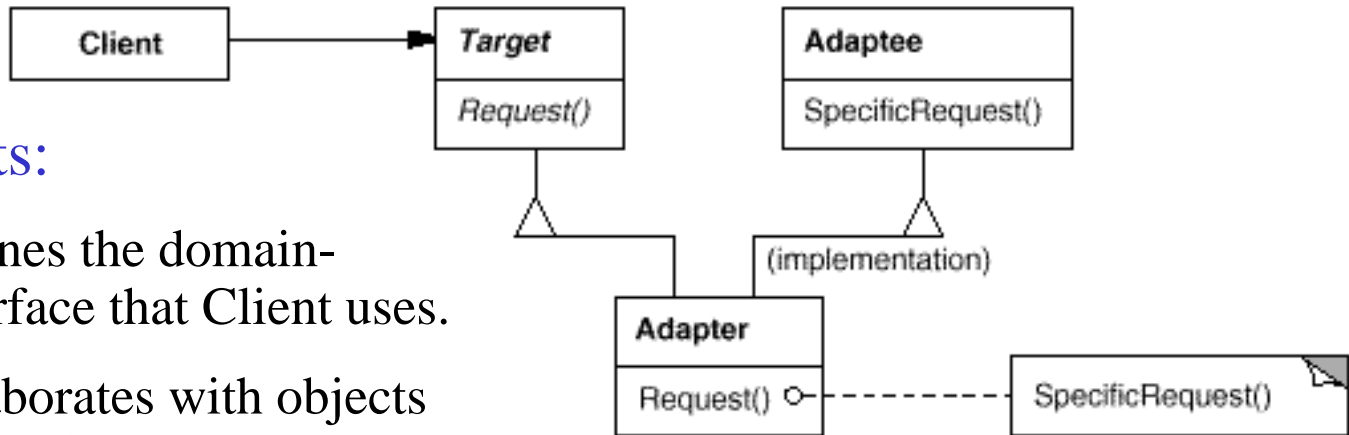
- you want to use an existing class, and its interface does not match the one you need.
- you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
- (*object adapter only*) you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.



# Adapter/2

## ■ Structure

- A class adapter uses multiple inheritance to adapt one interface to another:

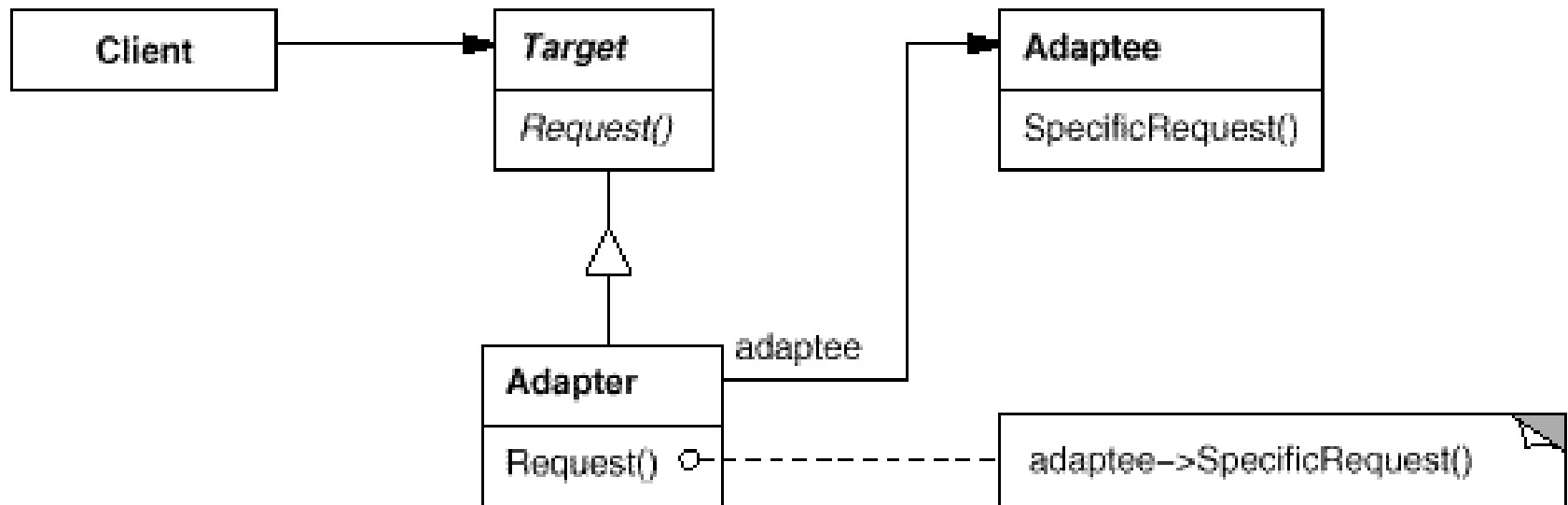


## ■ Participants:

- **Target:** defines the domain-specific interface that Client uses.
- **Client:** collaborates with objects conforming to the Target interface.
- **Adaptee:** defines interface that needs adapting.
- **Adapter:** adapts the interface of Adaptee to the Target interface.

# Adapter/2

- **Structure**
- **An object adapter relies on object composition:**



# Adapter/4

---

- **Consequences**
- Class and object adapters have different trade-offs.
- **A class adapter**
  - adapts Adaptee to Target by committing to a concrete Adapter class. A class adapter won't work when we want to adapt a class *and* all its subclasses.
  - lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
  - introduces only one object, and no additional pointer indirection is needed to get to the adaptee.
- **An object adapter**
  - lets a single Adapter work with many Adaptees, that is, the Adaptee itself and all of its subclasses (if any). The Adapter can also add functionality to all Adaptees at once.
  - makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

# Adapter/5

## ■ Example

- Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires. Consider for example a drawing editor that lets users draw and arrange graphical elements (lines, polygons, text, etc.) into pictures and diagrams. An user interface toolkit might already provide a sophisticated TextView class for displaying and editing text.

