Structural Design Patterns

Eduardo Figueiredo

http://www.dcc.ufmg.br/~figueiredo

Structural Patterns

- **Adapter**
- Bridge
- **Composite**
- **Decorator**
- Facade
- Flyweight
- Proxy

Adapter

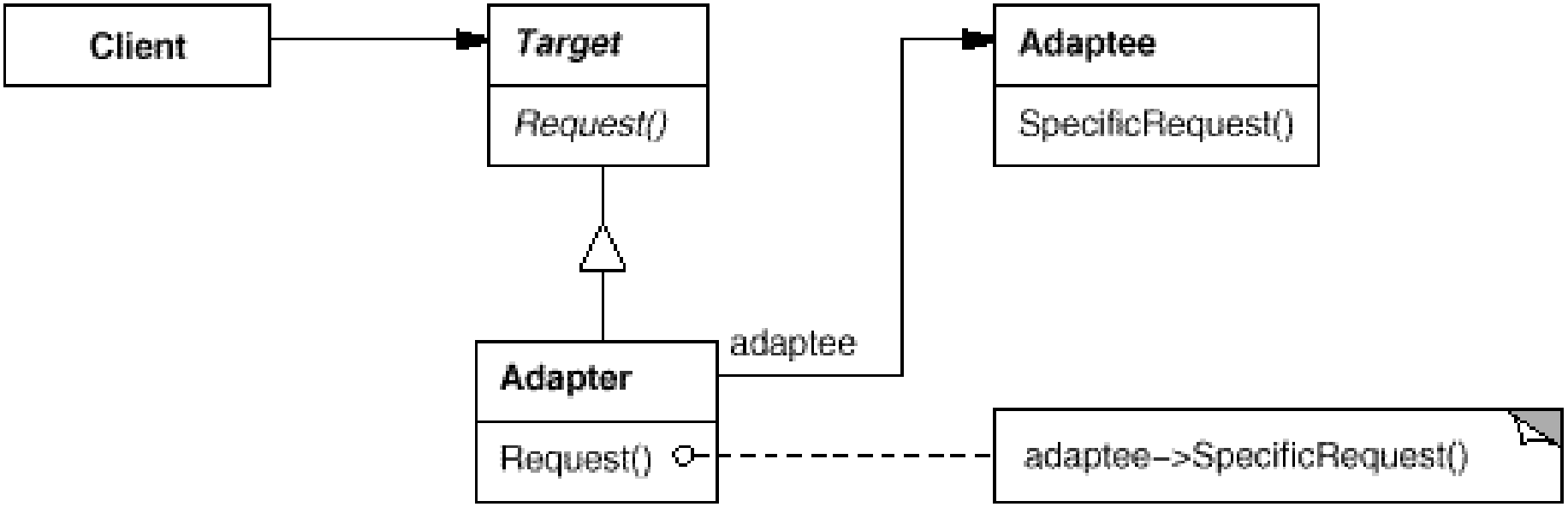
Motivation

- API is designed for reuse
  - It may not be reusable only because its interface (method signature) does not match the application interface
- We do not want (or can't) change the API interface
- We do not want (or can't) change the application interface

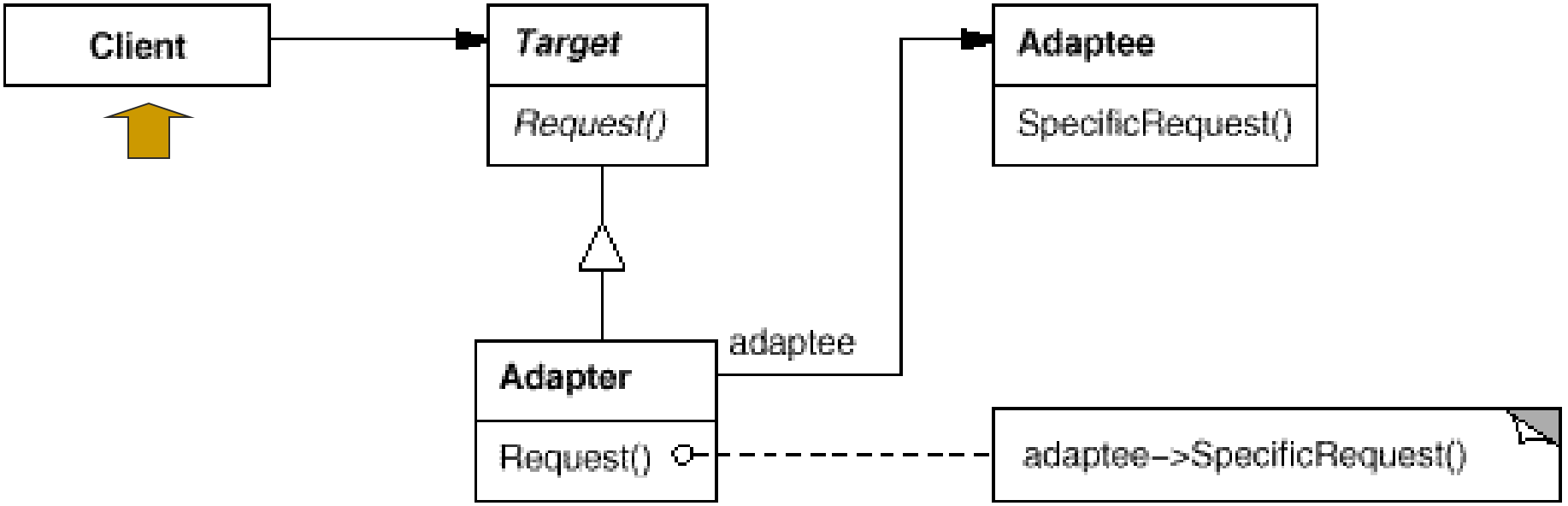
Adapter

- Name
  - Adapter
- Problem Description
  - Convert the interface of a class into another interface that clients expect
- Solution (*next slide*)
- Consequences
  - Support reuse without changing the API interface

Adapter Solution

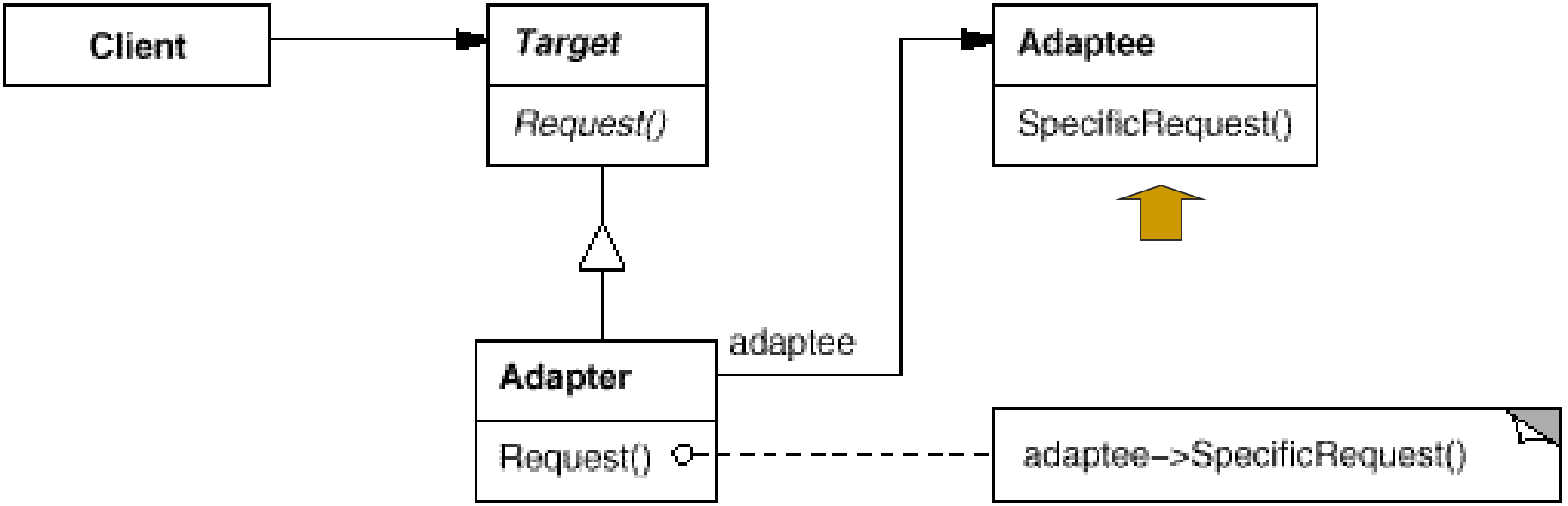


The Client Class



The client requires a complex functionality.

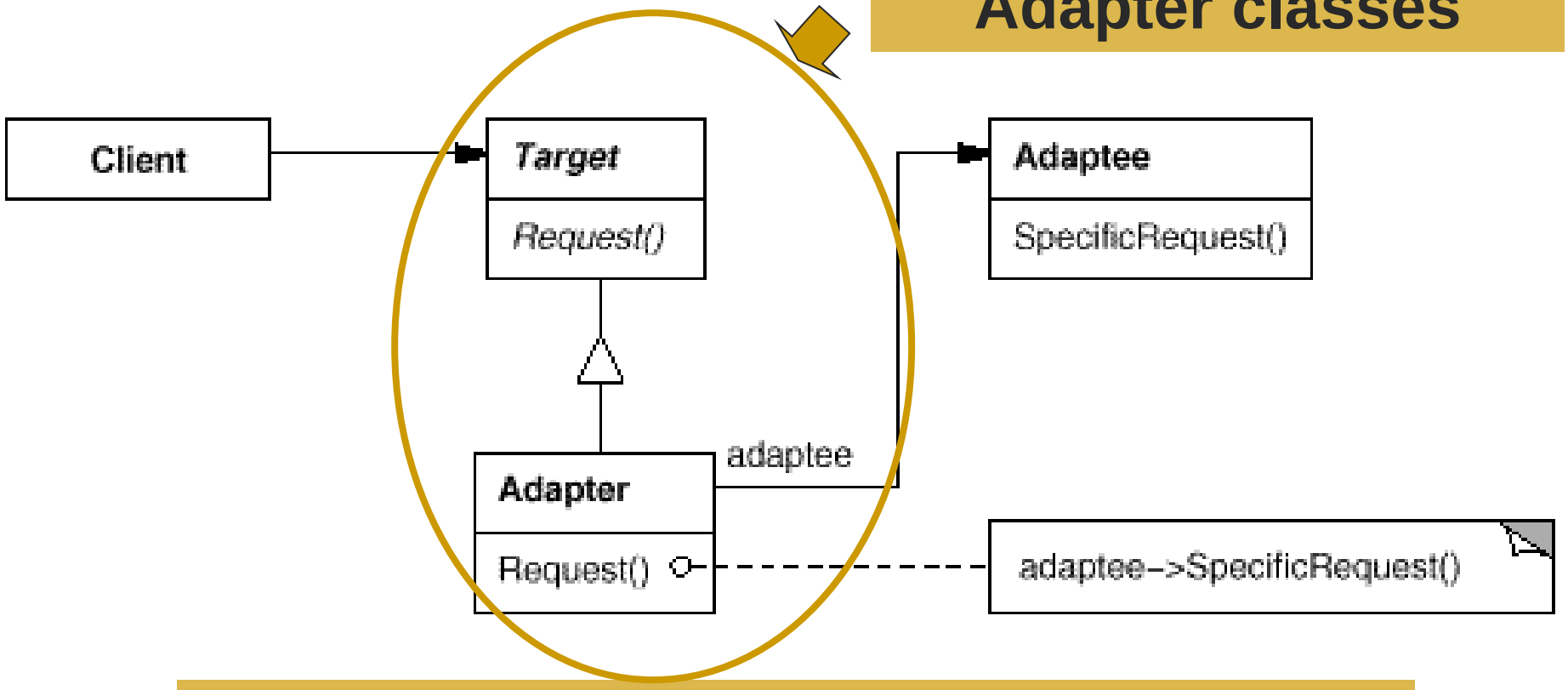
The Adaptee Class



The functionality is available in the Adaptee, but with a different signature.

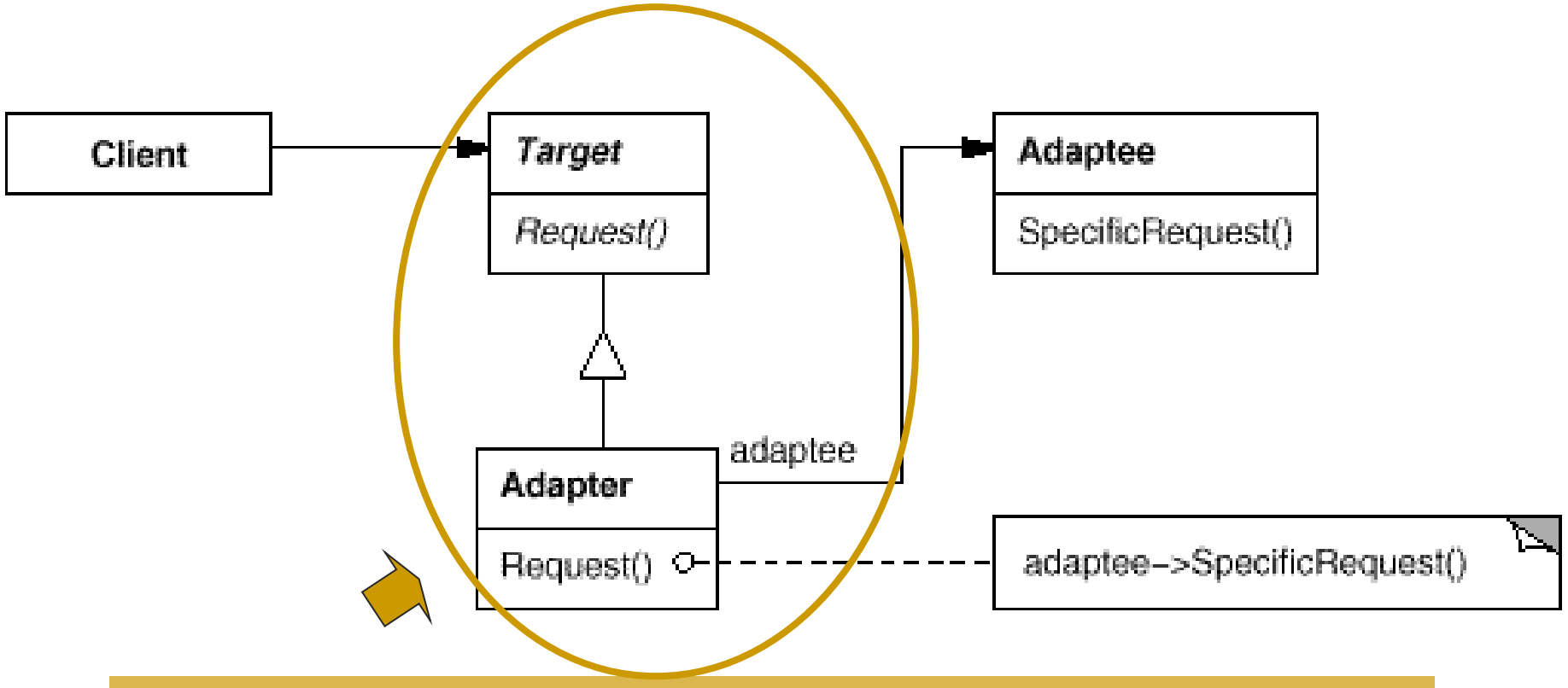
The Target Class

Adapter classes



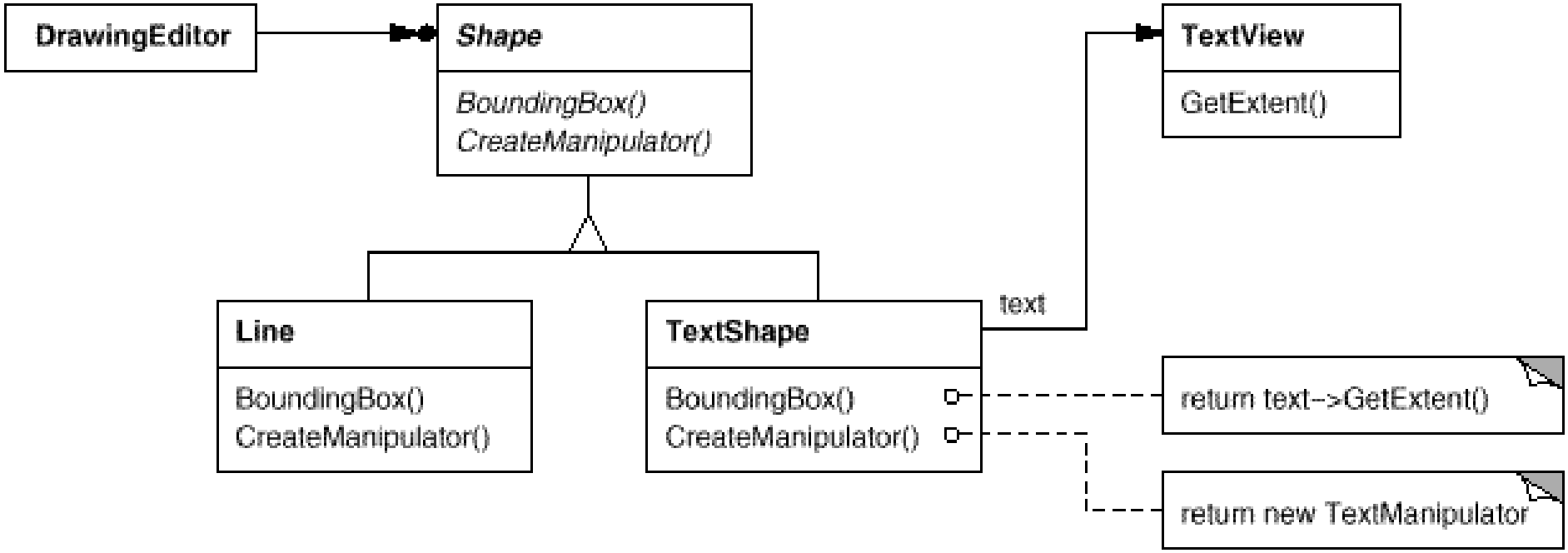
The *Target* class provides the method signature that the client requires.

The Adapter Class



The *Adapter* class changes the Adaptee interface into the client one.

Example of Use



***BoundingBox()* is the application method.**
***GetExtent()* is the equivalent API method.**

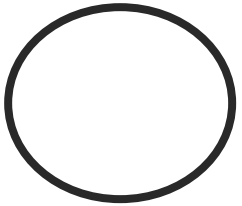
Composite

Motivating Example

- Some applications let users build complex diagrams out of simple components
- Example: Drawing Application

Simple Figures

line

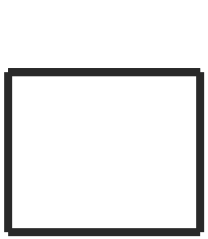


circle

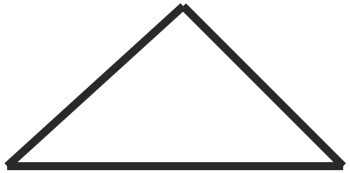
Motivating Example

- The user can group components to form larger components
  - Simple figures: line and circle

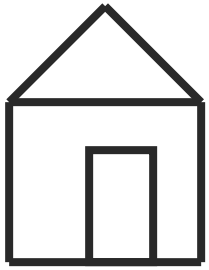
Complex Figures



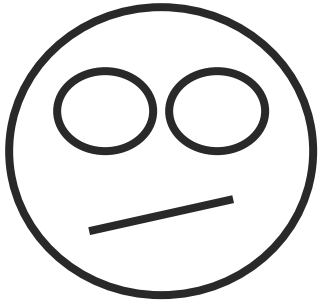
box



triangle

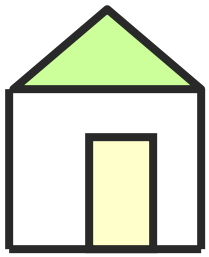


house



face

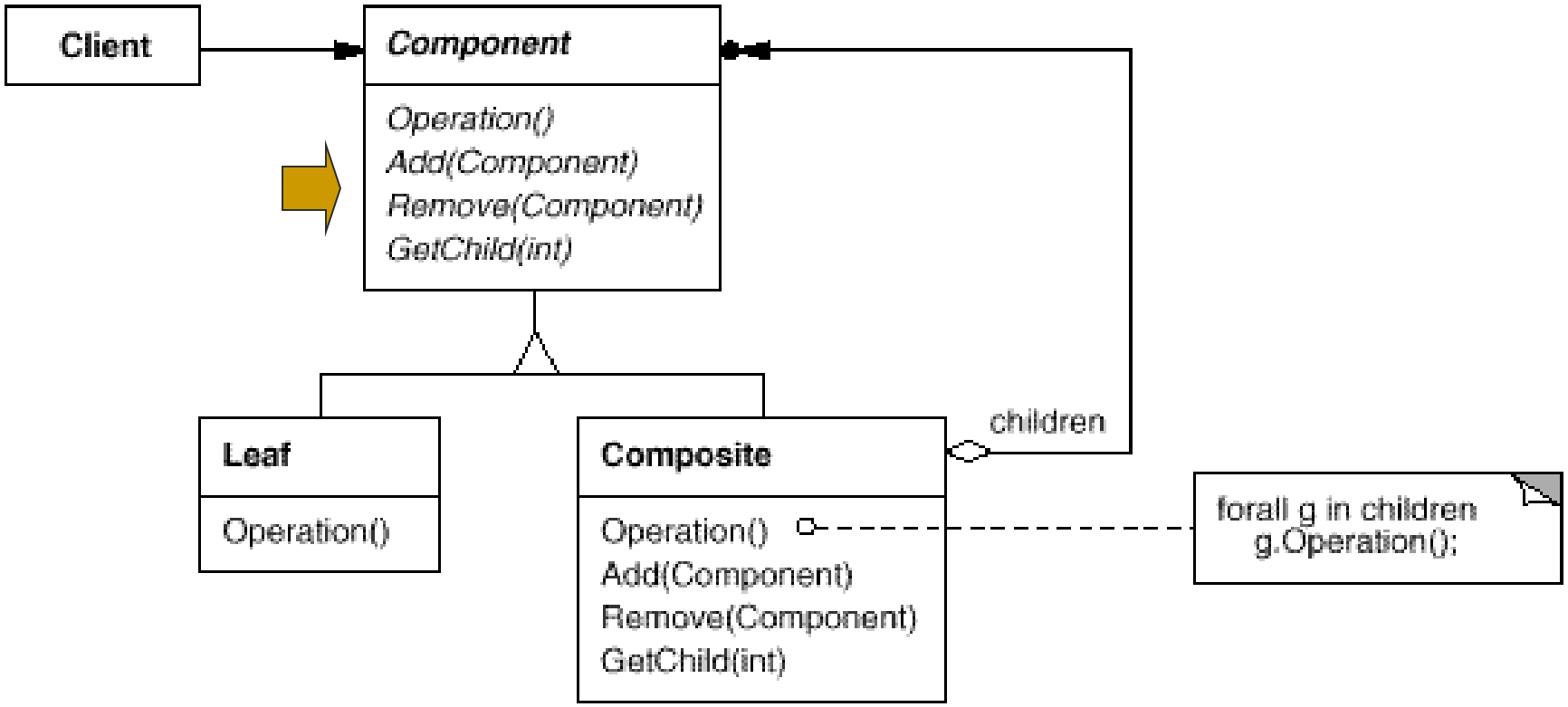
House Object



Composite

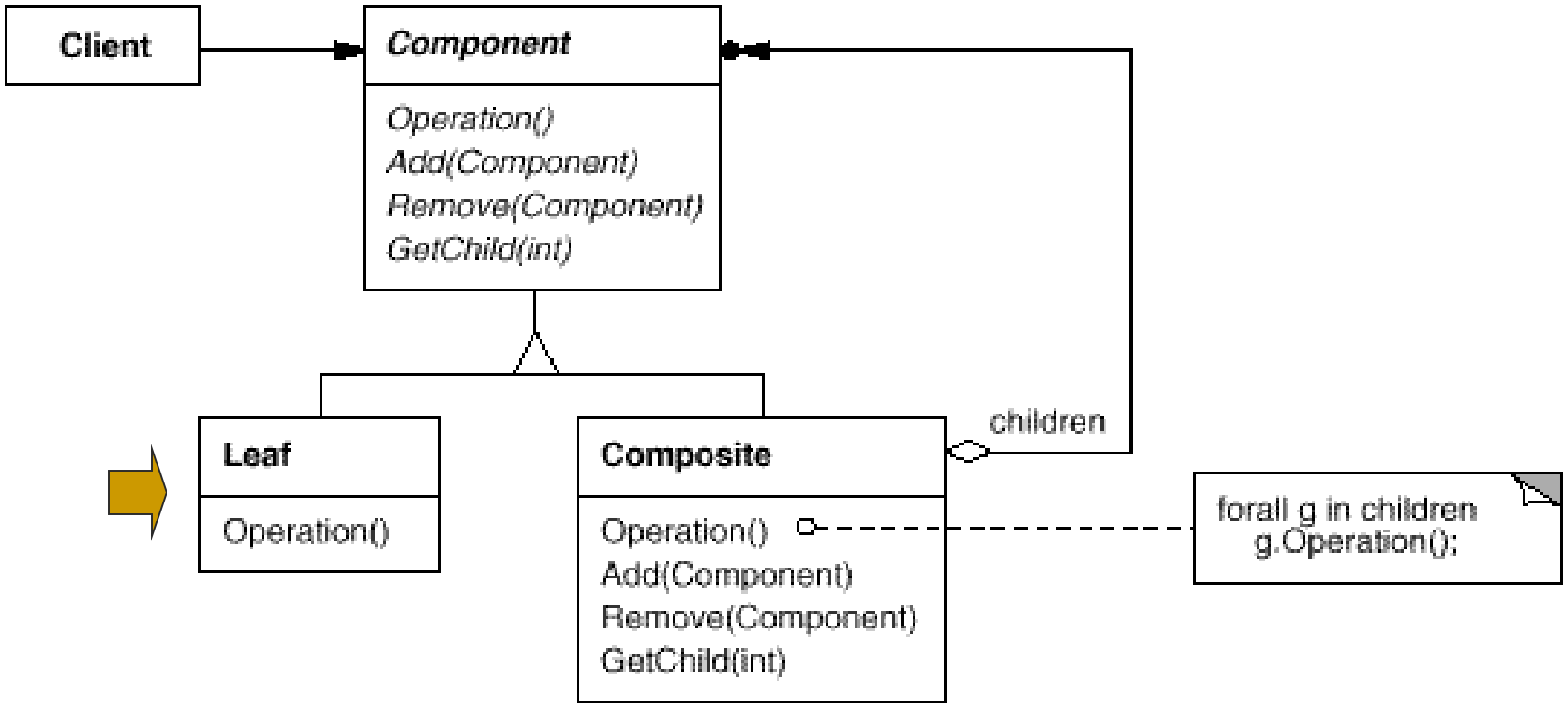
- Name
  - Composite
- Problem Description
  - Compose objects into tree structures to represent part-whole hierarchies
- Solution (*next slide*)
- Consequences
  - Clients will treat all objects uniformly
  - Makes it easier to add new components

Composite Solution



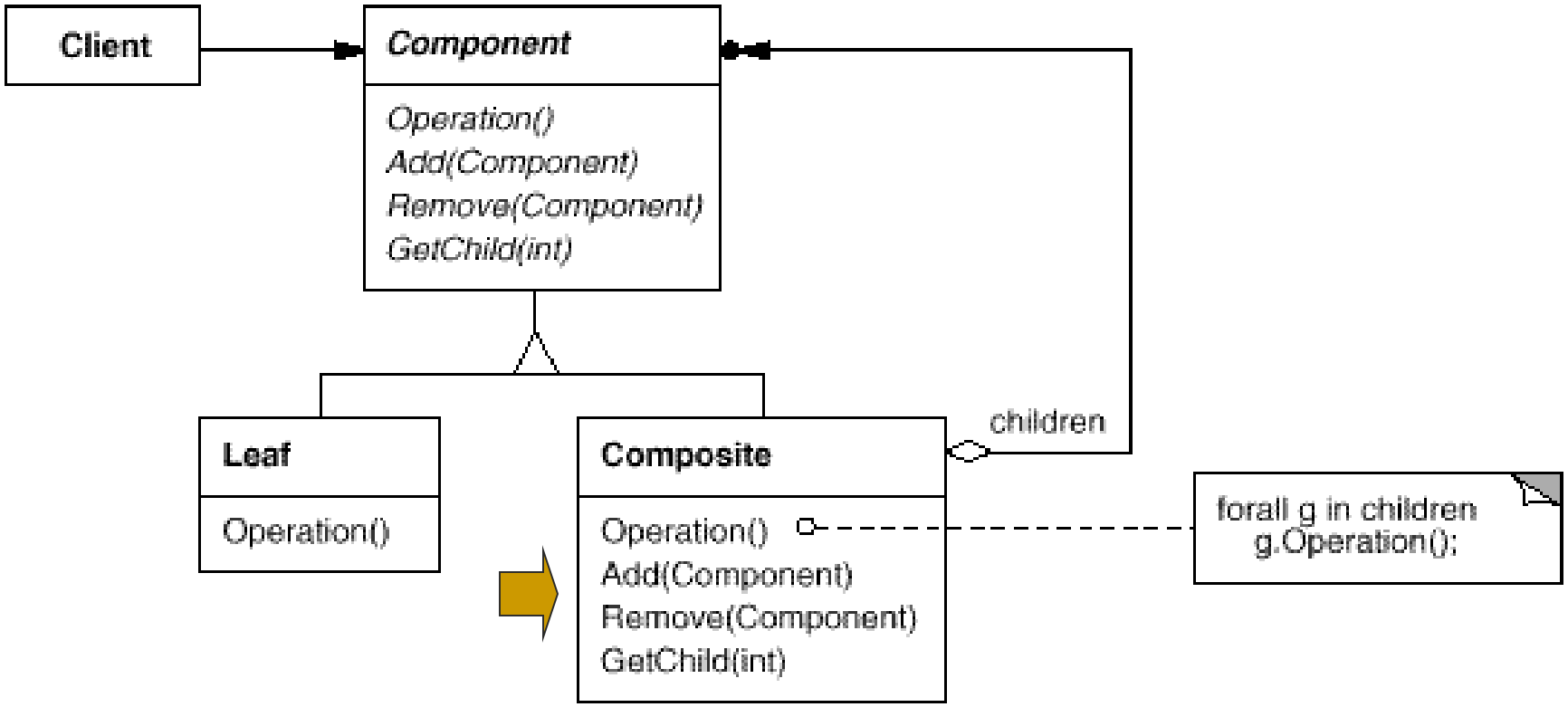
***Component* defines generic methods.**

The Leaf Class



***Leaf* overrides methods to give concrete implementations.**

The Composite Class



***Composite* represents complex objects.**

Decorator

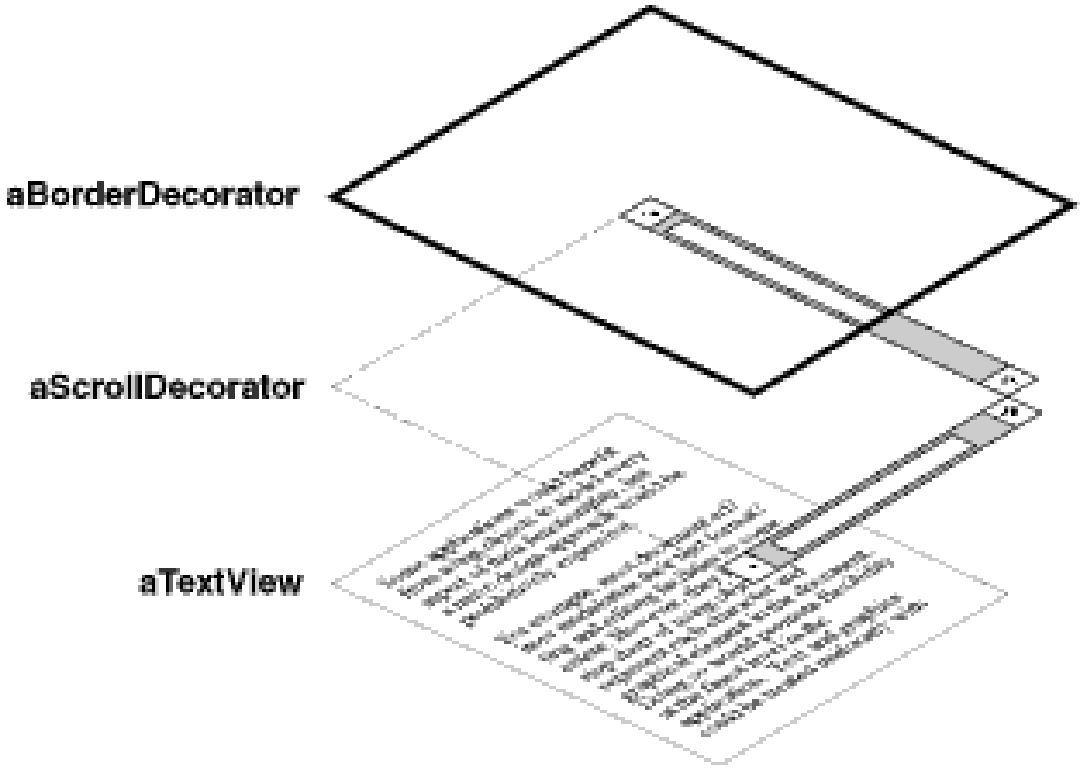
Motivation

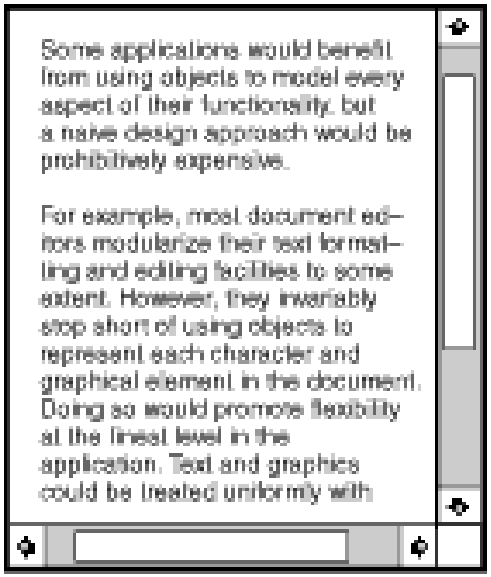
- We may want to add responsibilities to individual objects
  - But, not to an entire class

- One way to add responsibilities is with inheritance
  - Decorator is a smarter way of using inheritance

Motivating Example

- A graphical user interface toolkit
  - Borders and scrolling are dynamic properties

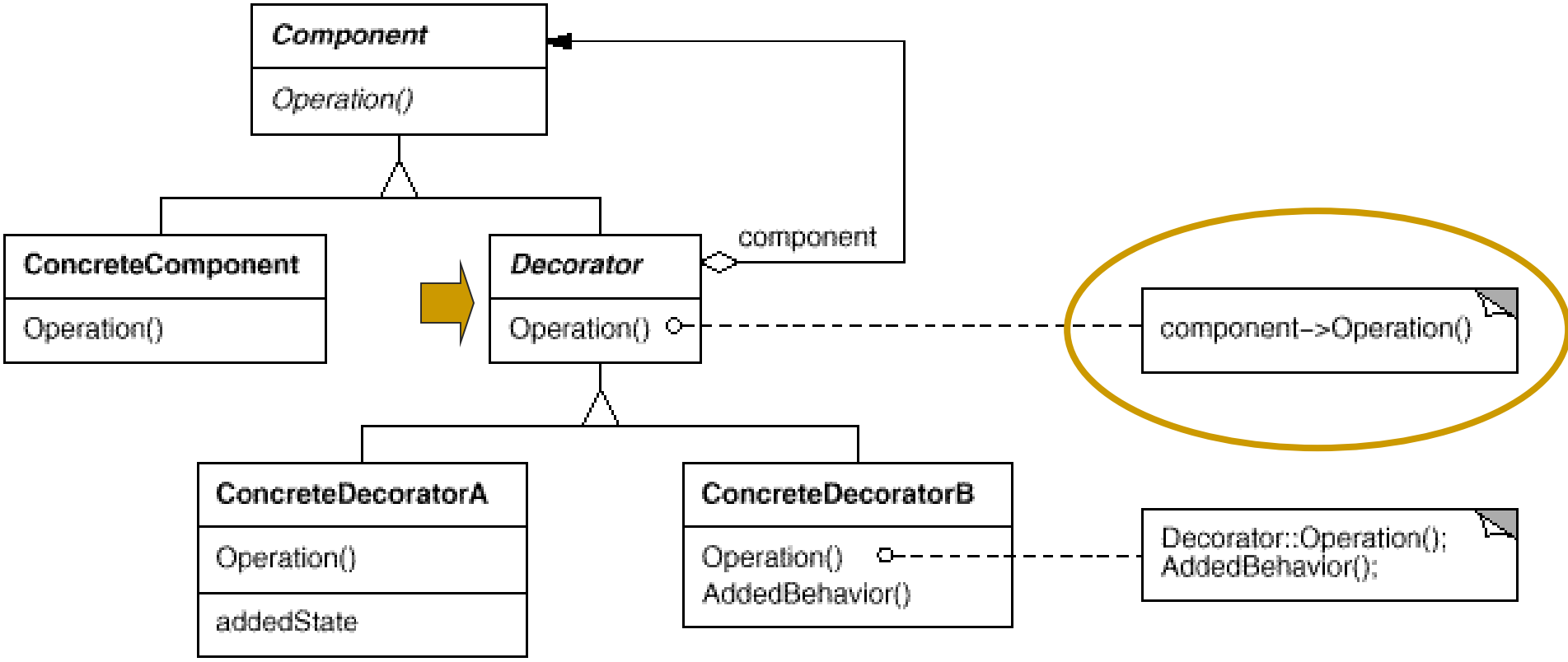




Decorator

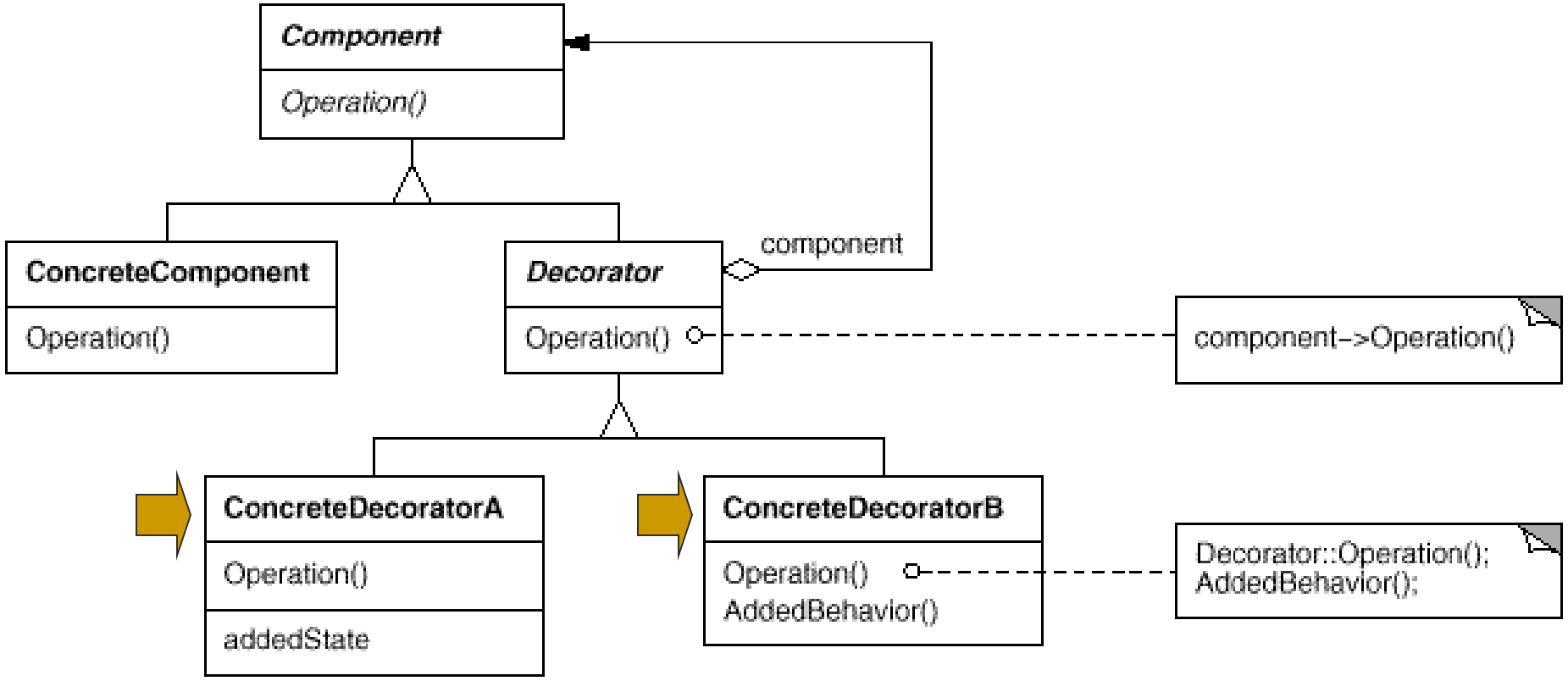
- Name
  - Decorator
- Problem Description
  - Attach additional responsibilities to an object dynamically
- Solution (*next slide*)
- Consequences
  - More flexibility than static inheritance
  - Responsibilities that can be withdrawn

Decorator Solution



A Decorator knows the component to decorate

As Classes ConcreteDecorator



Many concrete decorators are possible

Bibliography

- E. Gamma, R. Helm, R. Johnson, J. Vlissides. **Design Patterns**, 1st. Edition. Addison Wesley, 1994.
  - Adapter, Composite and Decorator