Design Defects and Restructuring

LECTURE 08

SAT, OCT 19, 2019

Structural Patterns

Adapter

Bridge

Composite

Decorator

Façade

Flyweight

Proxy

Adapter

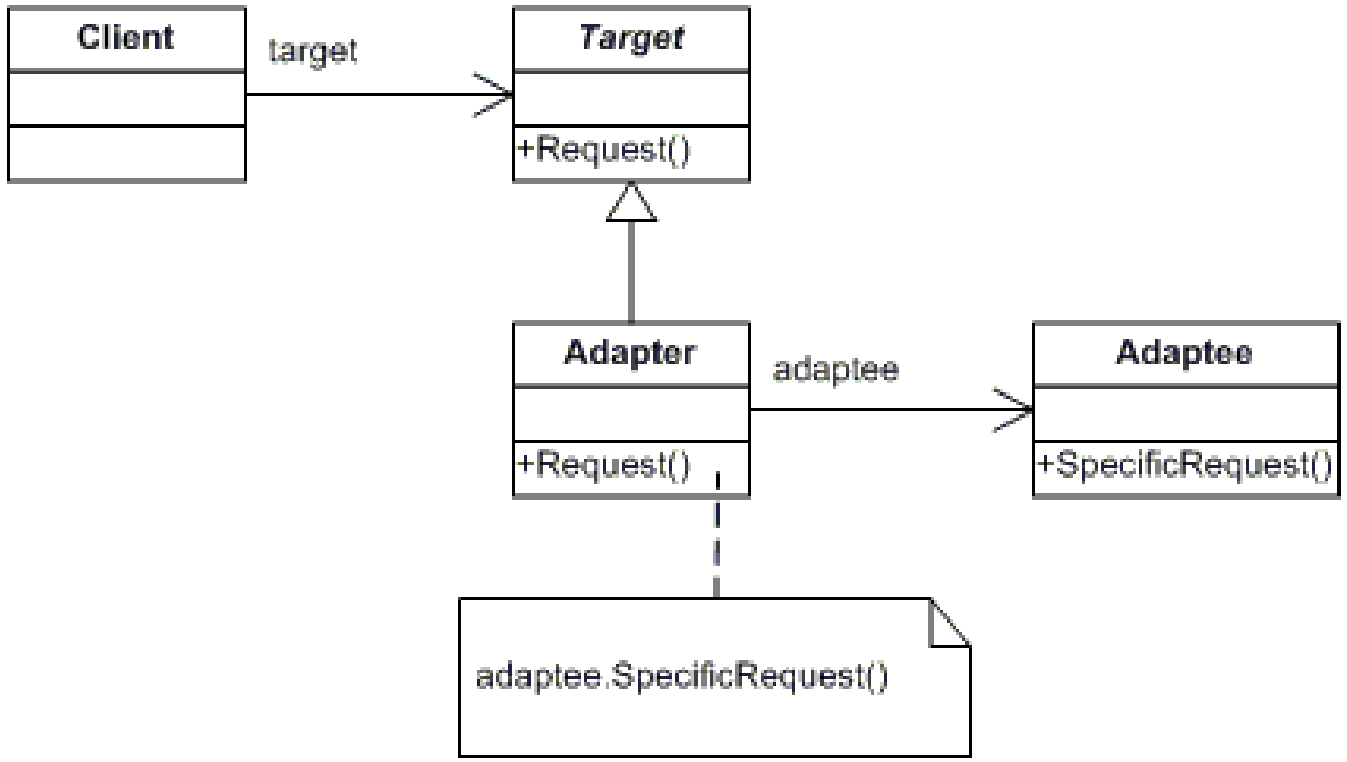
Intent

- Convert the interface of a class into another interface clients expect
- Adapter lets classes work together that could not 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 do not necessarily have compatible interfaces
- You need to use several existing subclasses, but it is impractical to adapt their interface by sub-classing every one
  - An object adapter can adapt the interface of its parent class

Adapter


Client
target
Target
+Request()
Adapter
+Request()
adaptee
Adaptee
+SpecificRequest()
adaptee.SpecificRequest()

Bridge

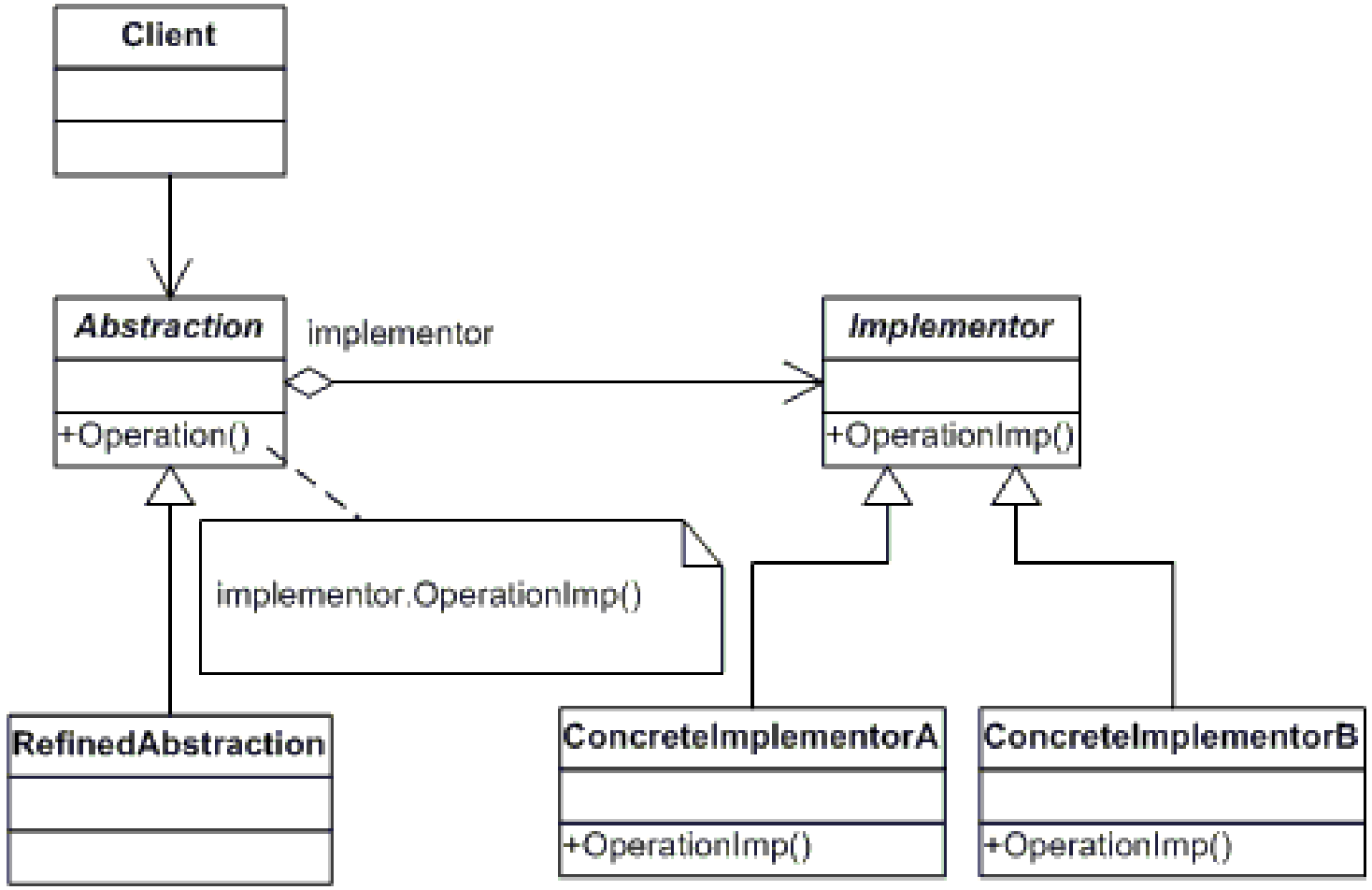
Intent

- Decouple an abstraction from its implementation so that the two can vary independently

Applicability

- You want to avoid a permanent binding between an abstraction and its implementation
- Both the abstractions and their implementations should be extensible by sub-classing
- Changes in the implementation of an abstraction should have no impact on clients

Bridge


Client
Abstraction
implementor
Implementor
+Operation()
+OperationImp()
implementor.OperationImp()
RefinedAbstraction
ConcreteImplementorA
+OperationImp()
ConcreteImplementorB
+OperationImp()

Composite

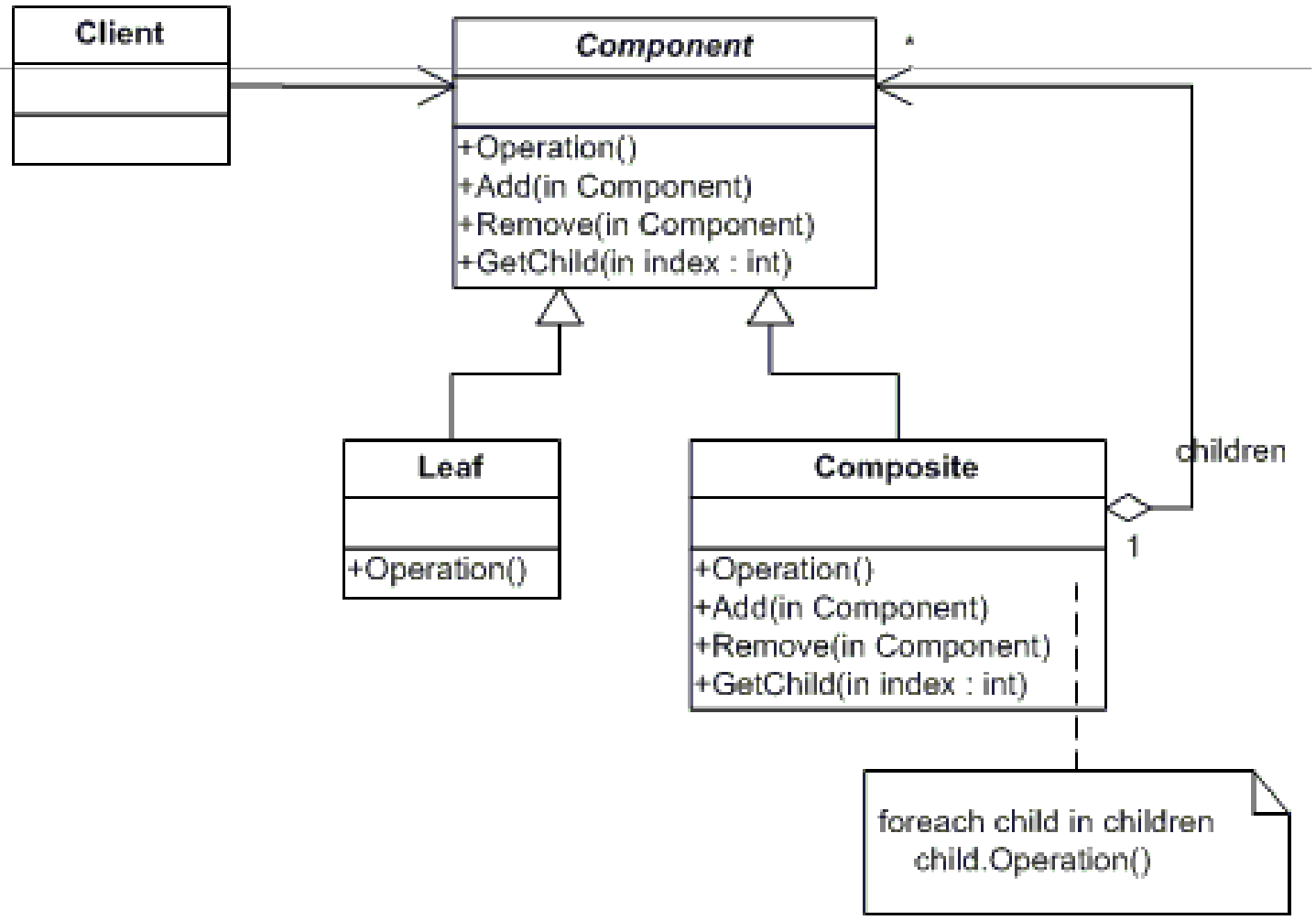
Intent

- 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


Client
Component
+Operation()
+Add(in Component)
+Remove(in Component)
+GetChild(in index : int)
*
Leaf
+Operation()
Composite
+Operation()
+Add(in Component)
+Remove(in Component)
+GetChild(in index : int)
children
1
foreach child in children
child.Operation()

Decorator

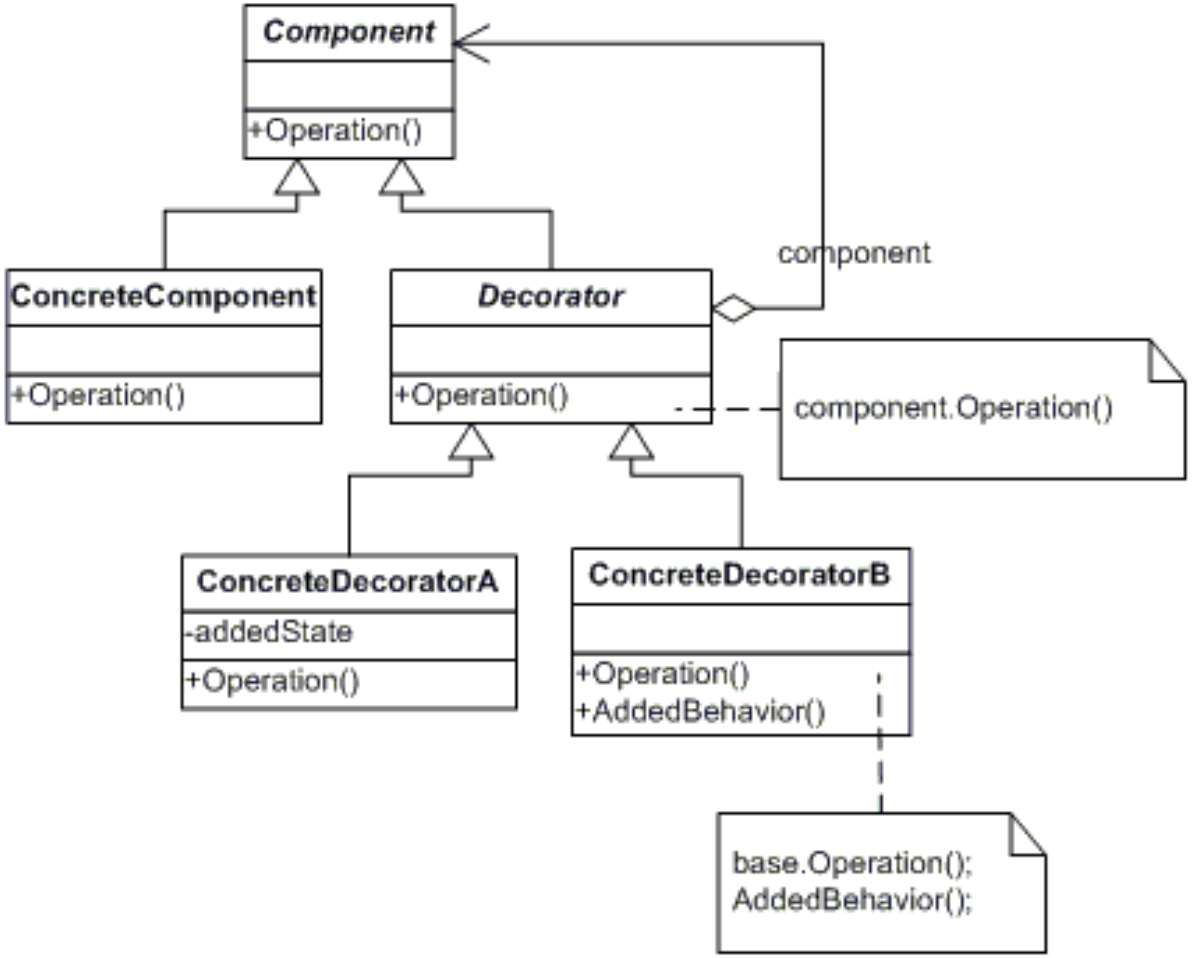
Intent

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

Applicability

- To add responsibilities to individual objects dynamically and transparently without affecting other objects
- For responsibilities that can be withdrawn
- When extension by sub-classing is impractical
  - Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination
  - Or a class definition may be hidden or otherwise unavailable for sub-classing

Decorator


Component
+Operation()
component
ConcreteComponent
+Operation()
Decorator
+Operation()
component.Operation()
ConcreteDecoratorA
-addedState
+Operation()
ConcreteDecoratorB
+Operation()
+AddedBehavior()
base.Operation();
AddedBehavior();