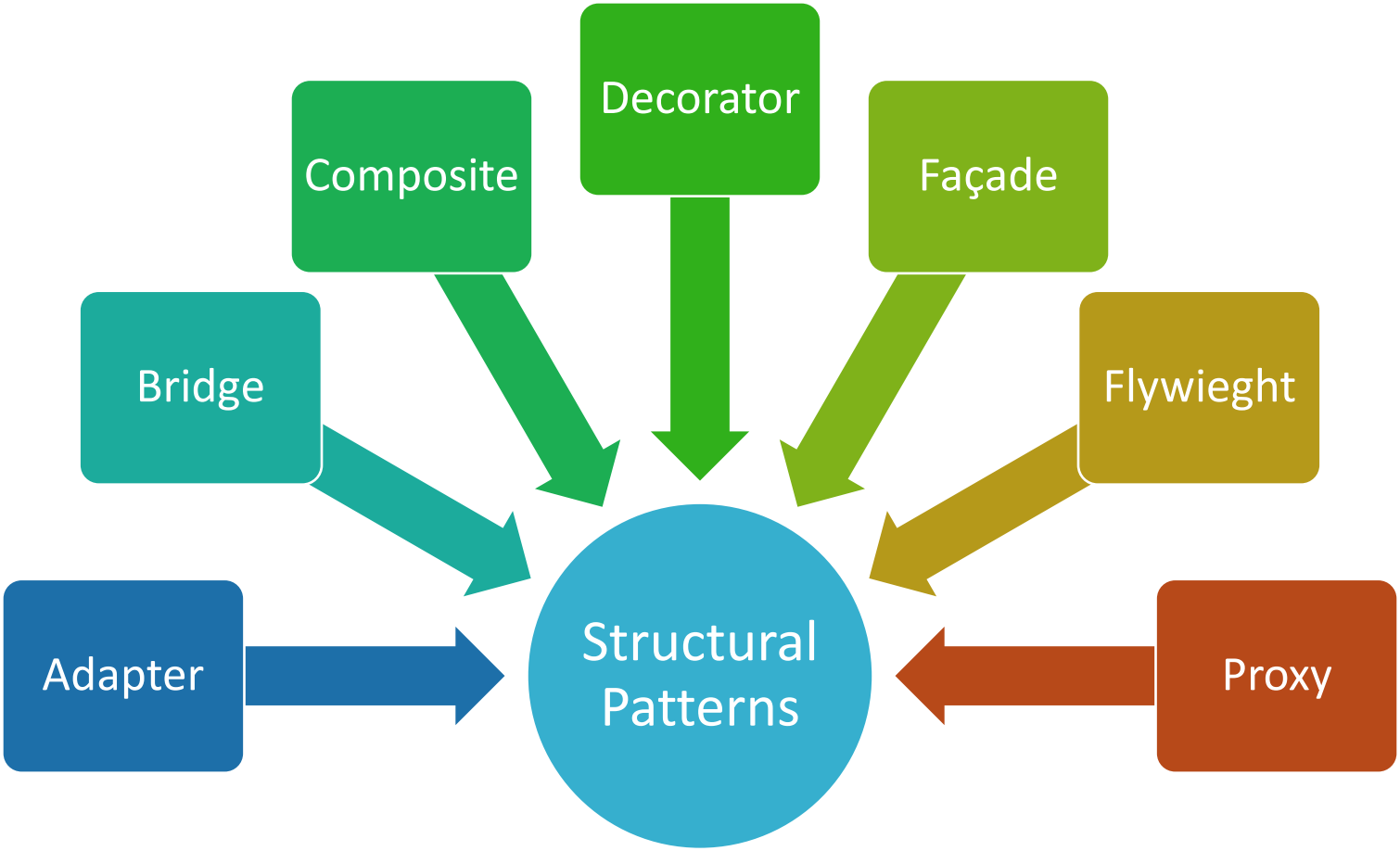
Design Patterns

Structural Patterns

Structural Patterns



Adapter Pattern

Adapter Pattern

Intent

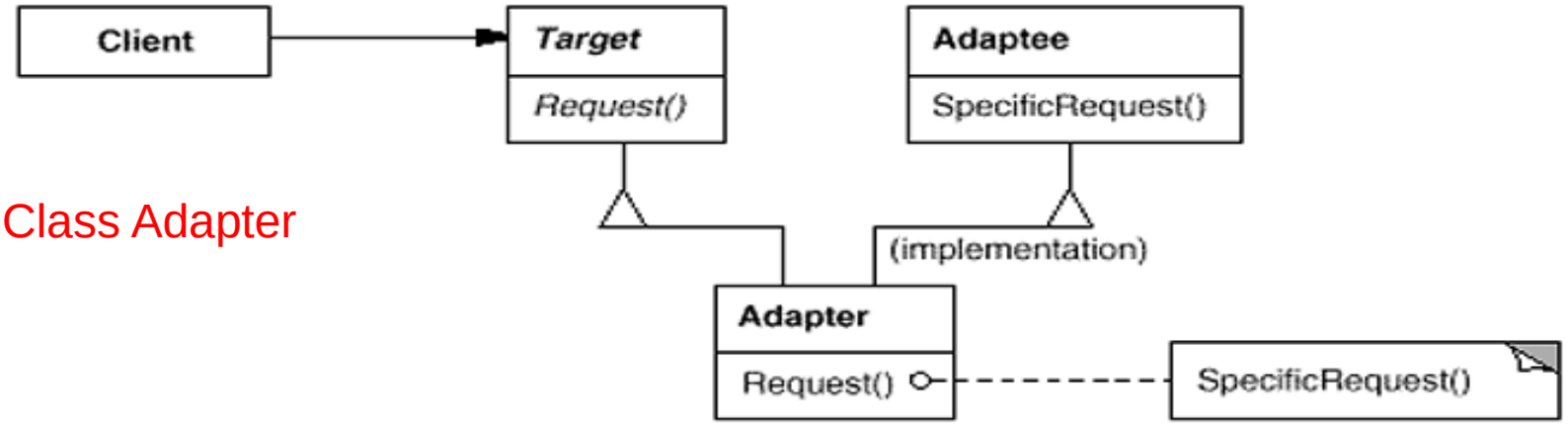
- Convert the interface of a class into another interface clients expect.
- Also known as **“Wrapper”**

Applicability

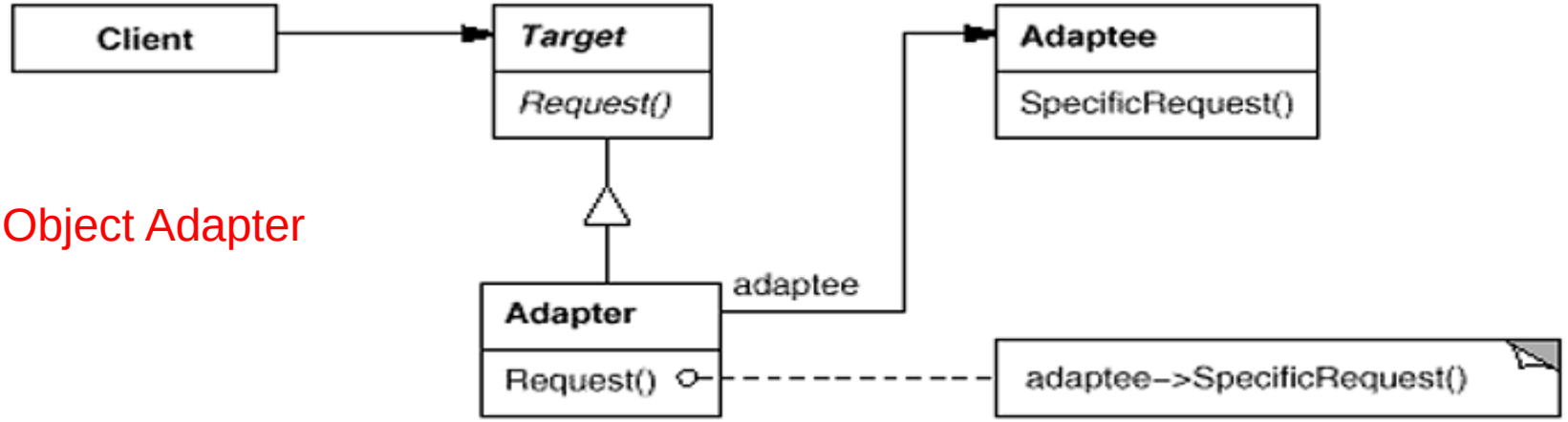
- When you want to use an existing class, and its interface does not match the one you need.
- Create a reusable class that cooperates with unrelated or unforeseen classes that don't necessarily have compatible interfaces,
- *(object adapter only)* Several existing subclasses are to be used, but it's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

Adapter - Structure

Class Adapter



Object Adapter



Adapter - Participants

Target	• Defines the domain-specific interface that Client uses
Client	• Collaborates with objects conforming to the Target interface.
Adaptee	• Defines an existing interface that needs adapting
Adapter	• Adapts the interface of Adaptee to the Target interface.

Adapter

Consequences

- Adapts Adaptee to Target by creating a concrete Adapter class
- Lets a single Adapter work with many Adaptees
- Using two-way adapters to provide transparency

Implementation

- Pluggable adapters
- Parameterized adapters

Demo

Bridge Pattern

Bridge Pattern

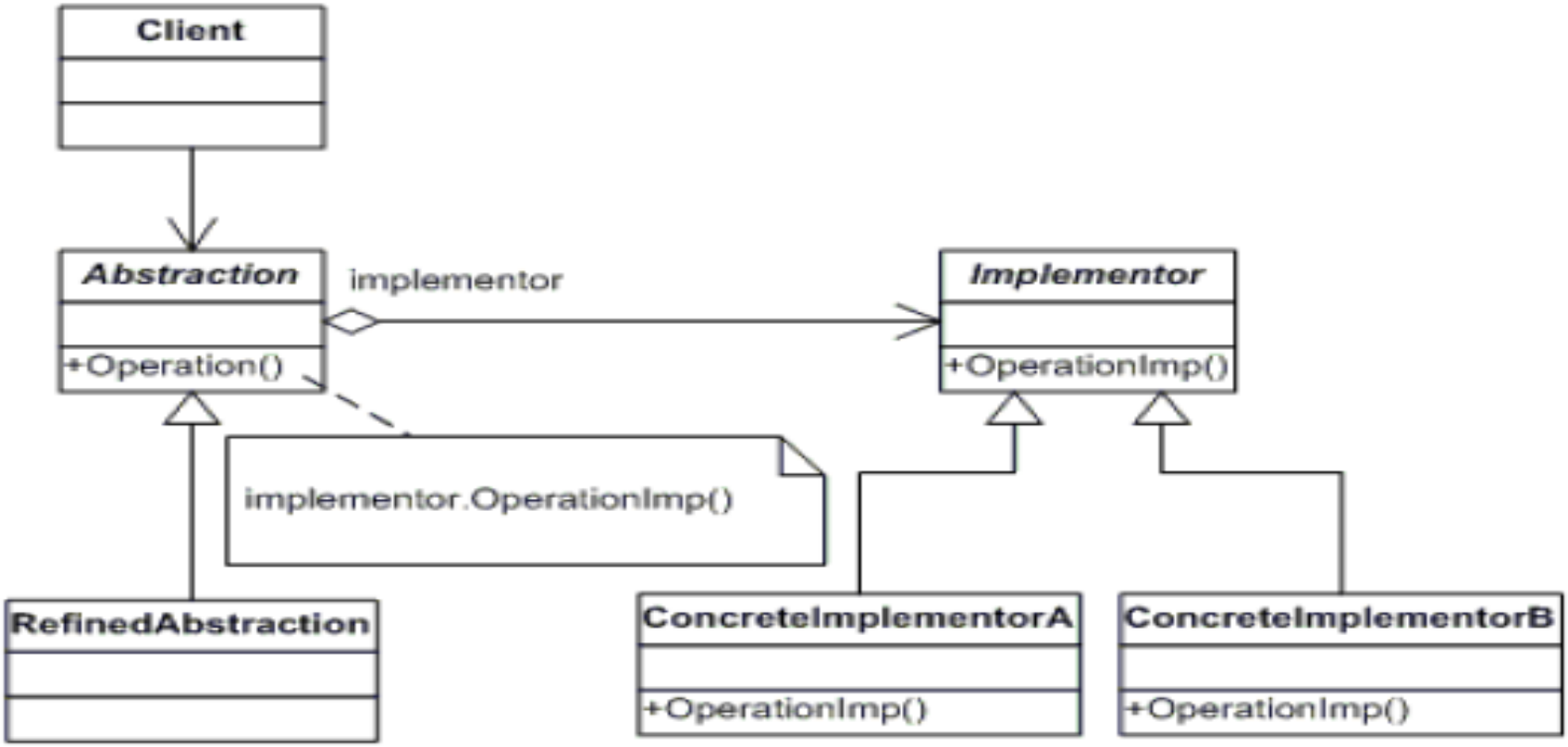
Intent

- Separates the interface of class from its implementation so that either can vary independently
- Also known as **“Handle/Body”**

Applicability

- Avoid a permanent binding between an abstraction and its implementation
- Both the abstractions and their implementations should be extensible by subclassing
- Changes in the implementation of an abstraction should have no impact on clients
- Hide the implementation of an abstraction completely from clients
- Shares an implementation among multiple objects

Bridge Pattern - Structure



Bridge Pattern - Participants

Abstraction	• Defines the abstraction's interface • Maintains a reference to an object of type Implementor
RefinedAbstraction	• Extends the interface defined by Abstraction.
Implementor	• defines the interface for implementation classes. • The Interface doesn't have to correspond exactly to Abstraction's interface
ConcreteImplementor	• Implements the Implementor interface and defines its concrete implementation.

Bridge Pattern

Consequences

- Decoupling interface and implementation
- Improved Extensibility
- Hide implementation details from client

Implementation

- Only one implementor
- Creating the right implementor object
- Sharing implementors
- Using Multiple inheritance

Demo

Composite Pattern

Composite Patterns

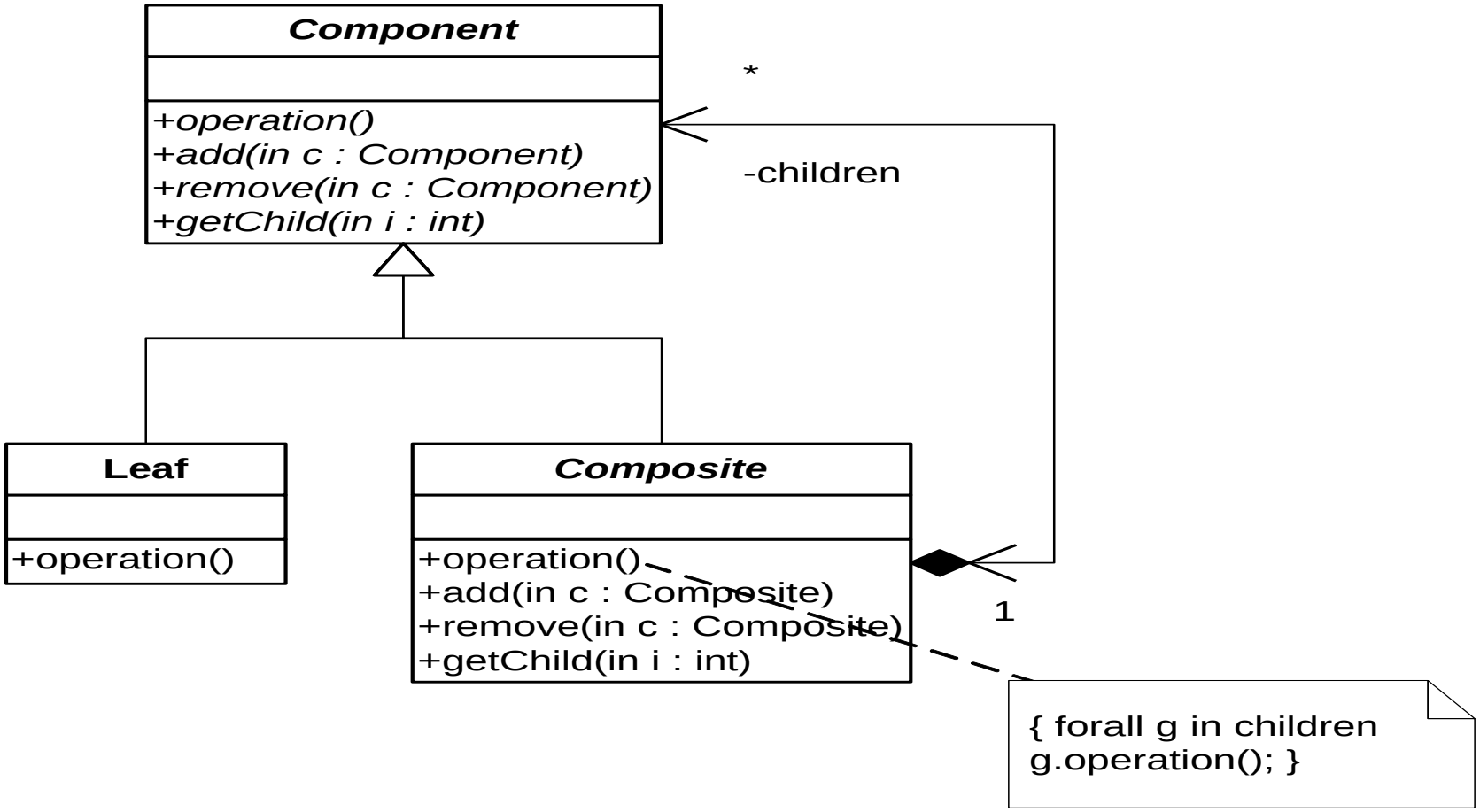
Intent

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

Applicability

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

Composite Pattern - Structure



Composite Pattern - Participants

Component	• Declares the interface for objects in the composition. • Implements default behavior for the interface common to all classes, as appropriate. • Declares an interface for accessing and managing its child components
Leaf	• Represents leaf objects in the composition. A leaf has no children. • Defines behavior for primitive objects in the composition.
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 Pattern

Consequences

- Defines class hierarchies consisting of primitive objects and composite objects
- Makes the client simple.
- Makes it easier to add new kinds of components
- Makes your design overly general.

Composite Pattern

Implementation

- Explicit parent references
- Sharing components
- Maximising the Component Interface
- Declaring the child management operations
- Should Component implement a list of Components ?
- Child Ordering
- Caching to improve performance
- Who should delete component?
- What’s the best data structure for storing components?

Demo

Decorator Pattern

Decorator Pattern

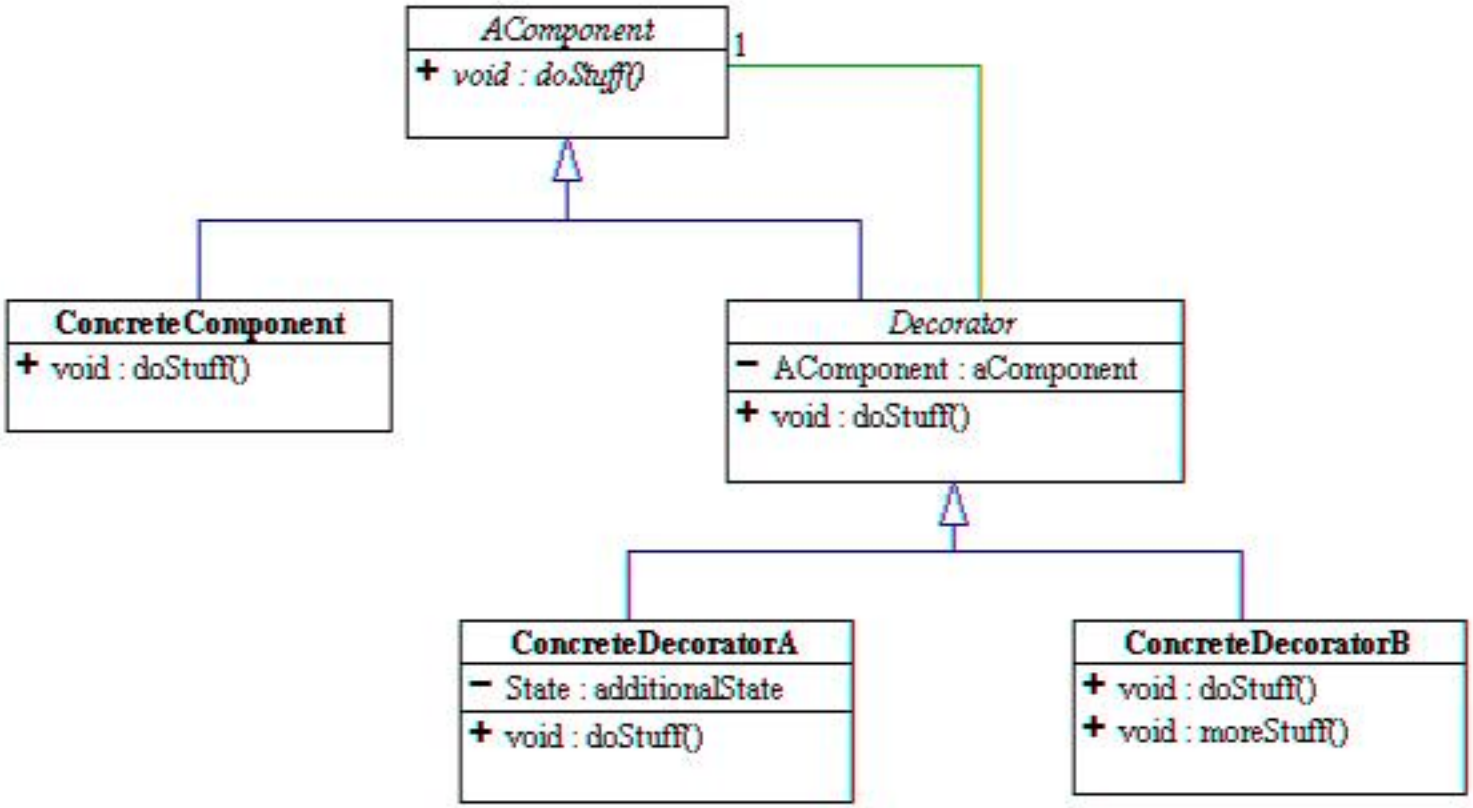
Intent

- Attach additional responsibilities to an object dynamically.
- Decorators provide a flexible alternative to subclassing for extending functionality
- Also known as **“Wrapper”**

Applicability

- Adds responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
- Responsibilities that can be withdrawn
- when extension by subclassing is impractical.

Decorator Pattern - Structure



Decorator Pattern - 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 Pattern

Consequences

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

Implementation

- Interface conformance
- Omit abstract Decorator class
- Keep Component classes lightweight
- Changing the skin of the object versus changing its guts

Demo

Façade Pattern

Façade Pattern

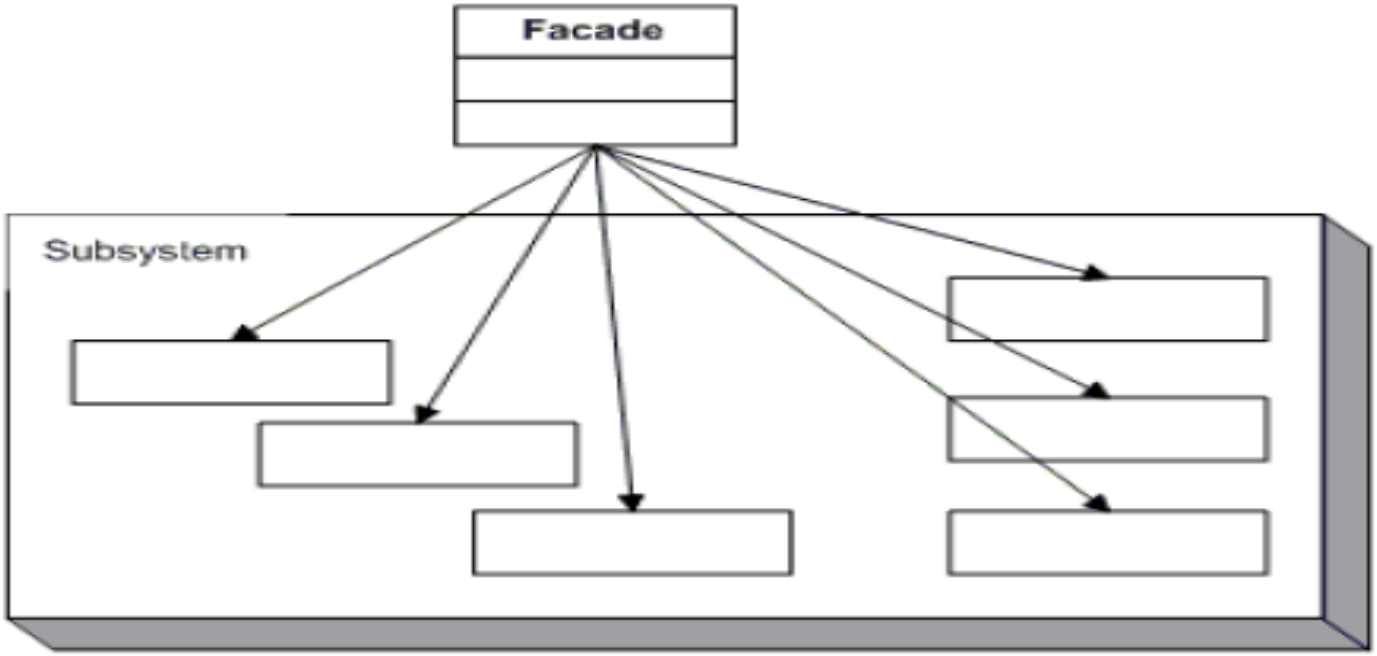
Intent

- Provides a unified interface to a set of objects in a subsystem.
- Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use

Applicability

- Provide a simple interface to a complex subsystem.
- Many dependencies between clients and the implementation classes of an abstraction.
- A Layer for the subsystems.
- Use a facade to define an entry point to each subsystem level.

Façade Pattern - Structure



Façade Pattern - Participants

Façade

- knows which subsystem classes are responsible for a request.
- delegates client requests to appropriate subsystem objects.

Subsystem classes

- implement subsystem functionality.
- handle work assigned by the Facade object.
- have no knowledge of the facade; that is, they keep no references to it.

Facade Pattern

Consequences

- Shields clients from subsystem components,
- Reduces the number of objects that clients deal with and making the subsystem easier to use
- Promotes weak coupling between the subsystem and its clients.
- Doesn’t prevent applications from using subsystem classes if they need to.

Implementation

- Reducing client-subsystem coupling
- Public versus private subsystem classes

Demo

Flyweight Pattern

Flyweight Pattern

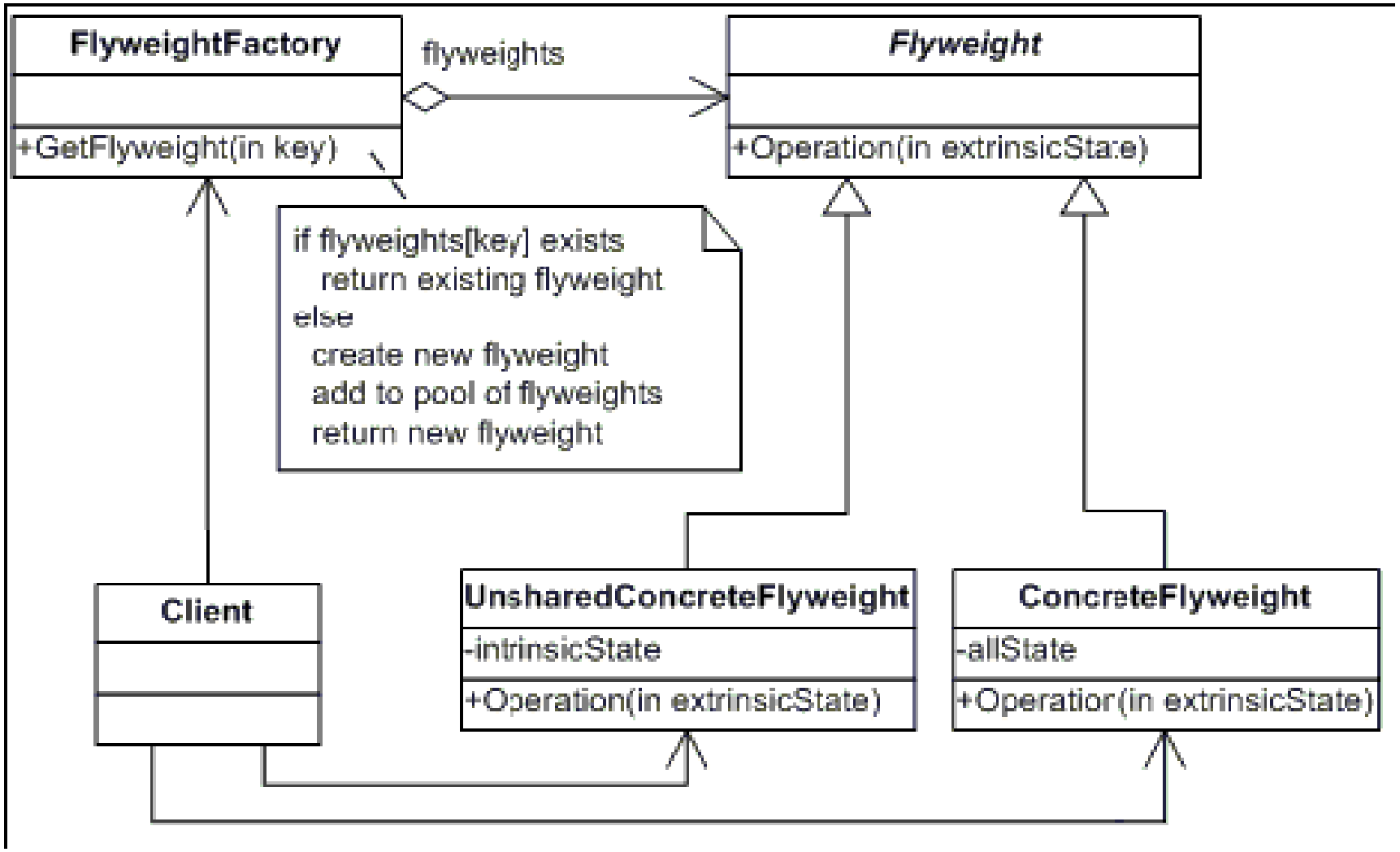
Intent

- Use sharing to support large numbers of fine-grained objects efficiently.

Applicability

- An application uses a large number of objects
- Storage costs are high because of sheer quantity of objects
- Most object state can be made extrinsic
- Many groups of objects can be replaced by relatively few shared objects once extrinsic state is removed
- The application doesn’t depend on object identity.

Flyweight Pattern - Structure



Flyweight Pattern - Participants

Flyweight	• declares an interface through which flyweights can receive and act on extrinsic state.
ConcreteFlyweight	• implements the Flyweight interface and adds storage for intrinsic state
UnsharedConcreteFlyweight	• not all Flyweight subclasses need to be shared
FlyweightFactory	• creates and manages flyweight objects. • ensures that flyweights are shared properly.
Client	• maintains a reference to flyweight(s). • computes or stores the extrinsic state of flyweight(s).

Flyweight Pattern

Consequences

- Flyweights might introduce run-time costs associated with transferring, finding and/or computing extrinsic state.
- Such costs are offset by space saving.
- The more the flyweight object shared, the more the space saved.
- Combine with Composite pattern to represent a hierarchical structure.

Implementation

- Removing extrinsic state
- Managing shared objects

Demo

Proxy Pattern

Proxy Pattern

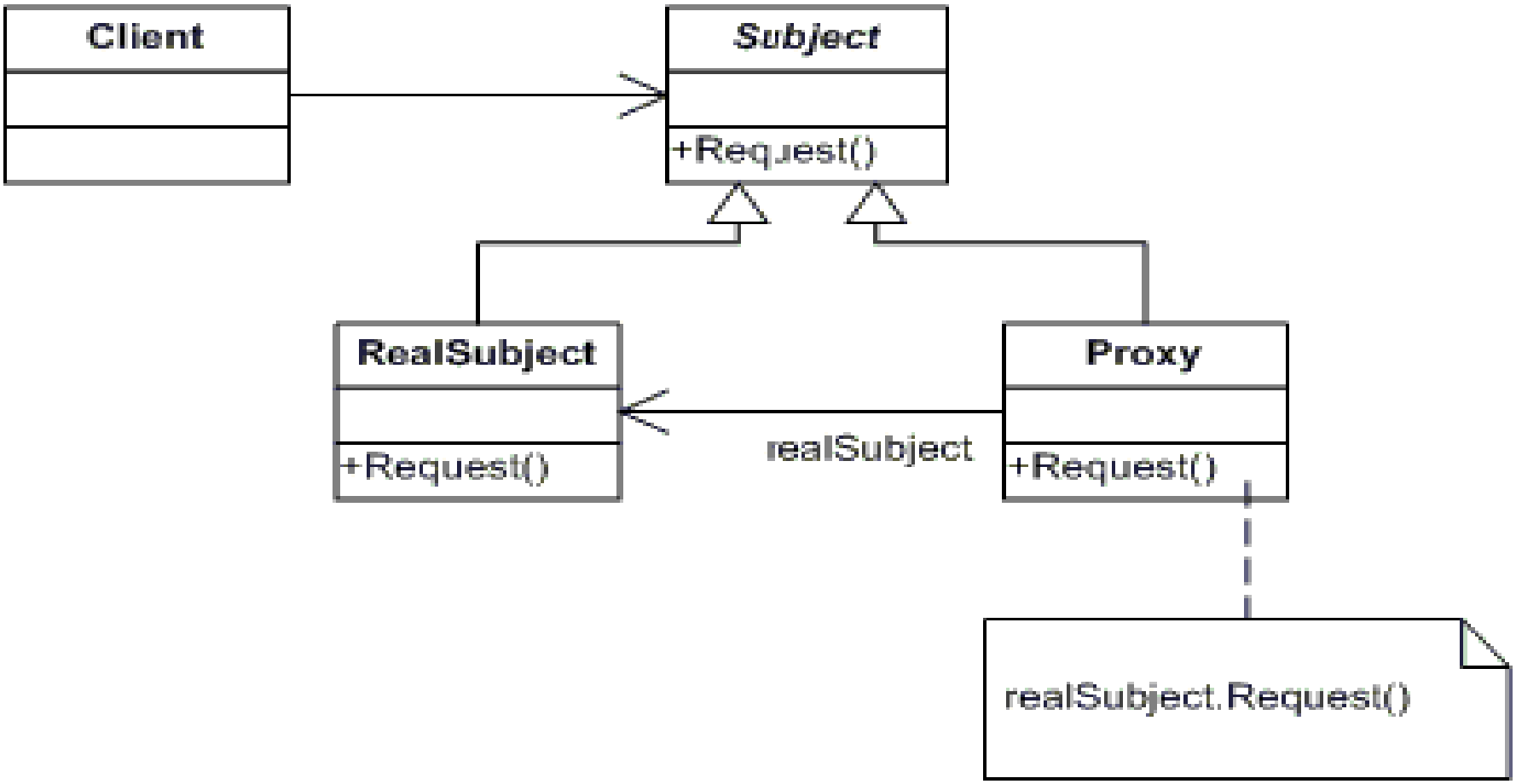
Intent

- Provide a surrogate or placeholder for another object to control access to it.
- Also known as **“Surrogate”.**

Applicability

- Whenever there is a need for a more versatile and sophisticated reference to an object than a simple pointer.
- Several scenarios where used or different types of proxies
  - Communication Proxy
  - Virtual Proxy
  - Protection Proxy
  - Property Proxy

Proxy Pattern - Structure



Proxy Pattern - Participants

Proxy

- Maintains a reference that lets the proxy access the real subject.
- Provides an interface identical to Subject's so that a proxy can by substituted for the real subject.
- Controls access to the real subject and may be responsible for creating and deleting it

Subject

- Defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.

RealSubject

- Defines the real object that the proxy represents.

Proxy Pattern

Consequences

- Remote Proxies hides the fact that an object resides in a different address space.
- Virtual Proxies can perform optimizations such as creating an object on demand
- Protection Proxy and Smart Reference - allow additional housekeeping tasks when as object is accessed.
- Supports copy-on-demand operations.

Implementation

- Overloading the member access operator
- Proxy doesn't always have to know the type of real subject

Demo