Syllabus: Structural Pattern: Adapter, Bridge, Composite, Decorator, facade, Flyweight, Proxy.

Structural Patterns

Structural design pattern is a blueprint of how different **objects** and **classes** are combined together to form a bigger structure for achieving multiple goals altogether. The patterns in structural designs show how unique pieces of a system can be combined together in an extensible and flexible manner. So, with the help **structural design pattern** we can target and change a specific part of the structure without changing the entire structure.

Types of structural design patterns

There are following 7 types of structural design patterns.

1. **Adapter Pattern**: Adapting an interface into another according to client expectation.
2. **Bridge Pattern**: Separating abstraction (interface) from implementation.
3. **Composite Pattern**: Allowing clients to operate on hierarchy of objects.
4. **Decorator Pattern**: Adding functionality to an object dynamically.
5. **Facade Pattern**: Providing an interface to a set of interfaces.
6. **Flyweight Pattern**: Reusing an object by sharing it.
7. **Proxy Pattern**: Representing another object.

Adapter Pattern

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.

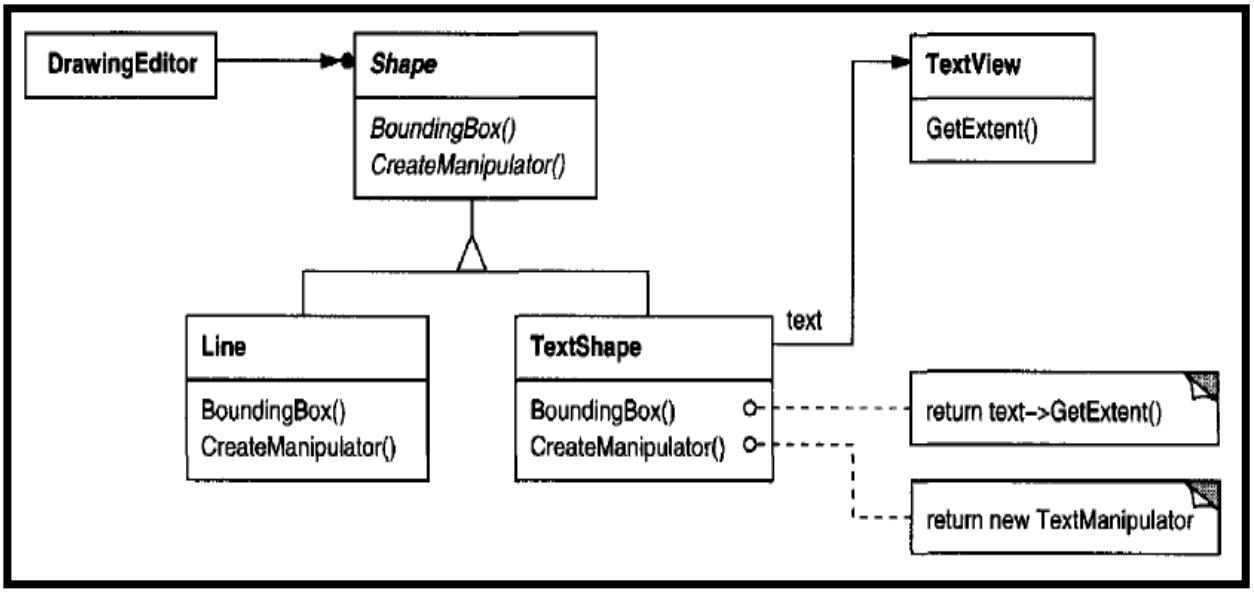
Also Known As

Wrapper

Motivation

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. The drawing editor's key abstraction is the graphical object, which has an editable shape and can draw itself. The interface for graphical objects is defined by an abstract class called Shape. The editor defines a subclass of Shape for each kind of graphical object: a LineShape class for lines, a PolygonShape class for polygons, and so forth.



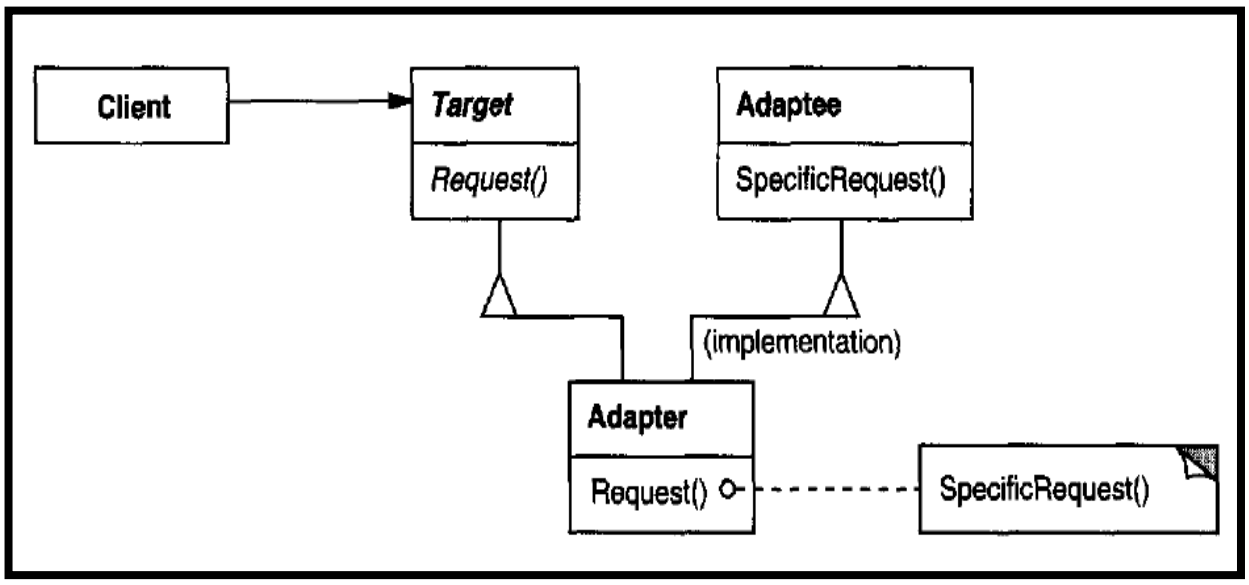
Applicability

Use the Adapter pattern when

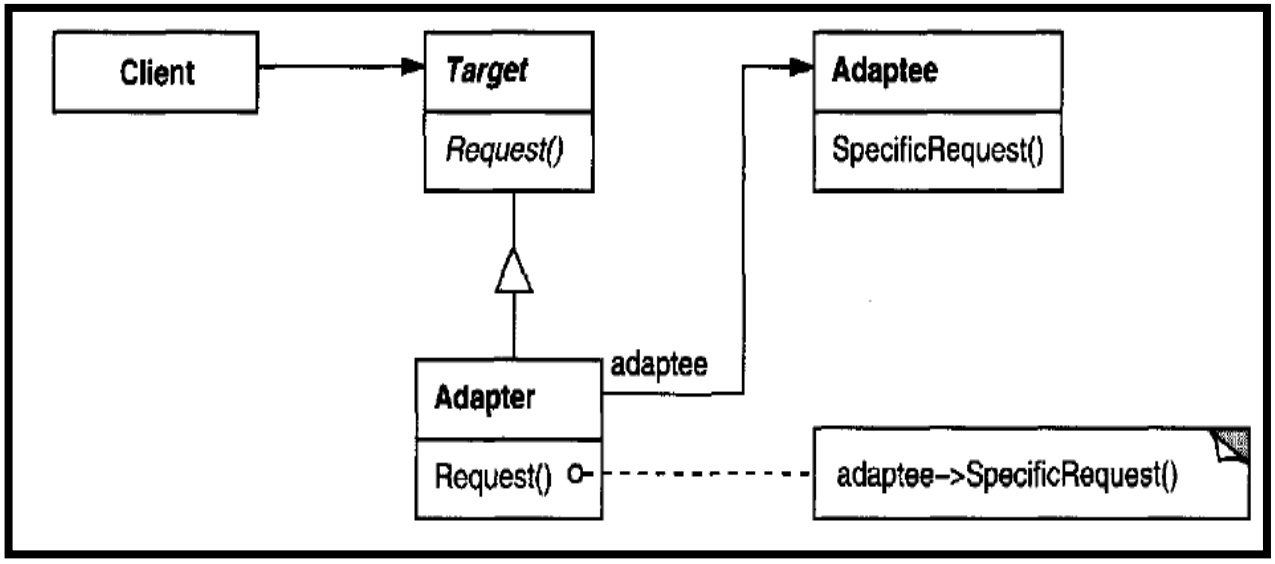
- ✓ You want to use an existing class, and its interface does not match the one you need.
- ✓ You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.

Structure

A class adapter uses multiple inheritances to adapt one interface to another:



An object adapter relies on object composition:



Participants

- ✓ **Target** (Shape): defines the domain-specific interface that Client uses.
- ✓ **Client** (DrawingEditor): collaborates with objects conforming to the Target interface.
- ✓ **Adaptec** (TextView): Defines an existing interface that needs adapting.
- ✓ **Adapter** (TextShape): Adapts the interface of Adaptec to the Target interface.

Collaborations

Clients call operations on an Adapter instance. In turn, the adapter calls Adaptec operations that carry out the request.

Consequences

Class and object adapters have different trade-offs. A class adapter

- ✓ Adapts Adaptee to Target by committing to a concrete Adaptee class. As a consequence, 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.

Implementation

Although the implementation of Adapter is usually straightforward, here are some issues to keep in mind:

Implementing class adapters in C++:

In a C++ implementation of a class adapter, Adapter would inherit publicly from Target and privately from Adaptec. Thus Adapter would be a subtype of Target but not of Adaptee.

Pluggable adapters:

Let's look at three ways to implement pluggable adapters for the TreeDisplay widget described earlier, which can lay out and display a hierarchical structure automatically.

Sample Code

We'll give a brief sketch of the implementation of class and object adapters for the Motivation example beginning with the classes Shape and TextView.

```
Class Shape
{
    public:
    Shape ();
    virtual void BoundingBox(
    Point& bottomLeft, Point& topRight
) const;
virtual Manipulator* CreateManipulator () const;
};
Class TextView
{
        public:
        TextView();

        void GetOrigin(Coord& x, Coord& y) const;

        void GetExtent(Coord& width, Coord& height) const;

        virtual bool IsEmpty() const;

};
```

Known Uses

The Motivation example comes from ET++Draw, a drawing application based on ET++.

Related Patterns

- Bridge has a structure similar to an object adapter, but Bridge has a different intent.
- Decorator enhances another object without changing its interface. A decorator is thus more transparent to the application than an adapter is.

Bridge Pattern

Intent

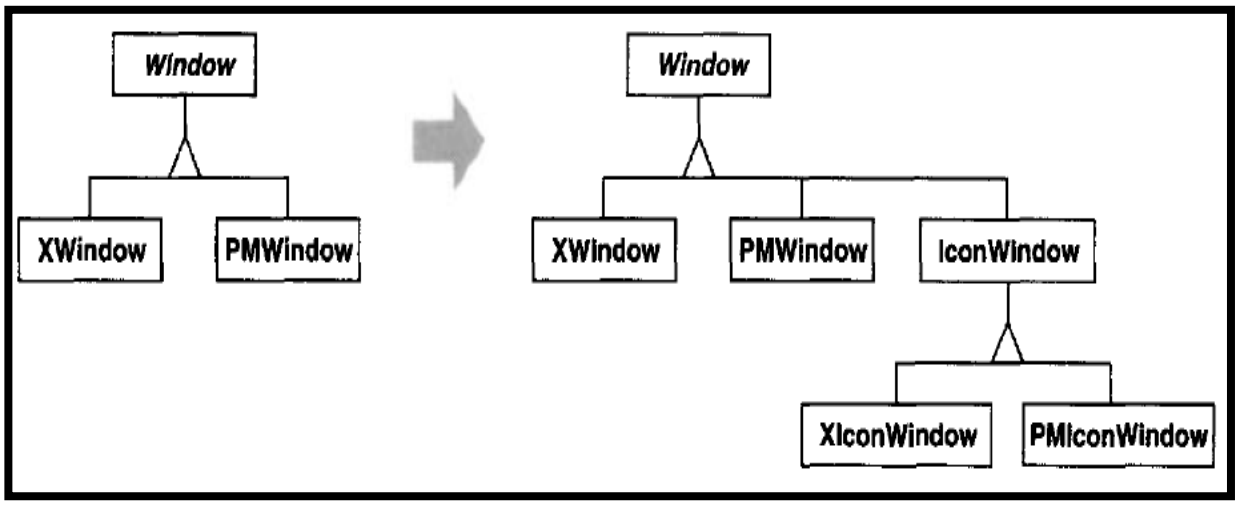
Decouple an abstraction from its implementation so that the two can vary independently.

Also Known As

Handle/Body

Motivation

When an abstraction can have one of several possible implementations, the usual way to accommodate them is to use inheritance.



Applicability

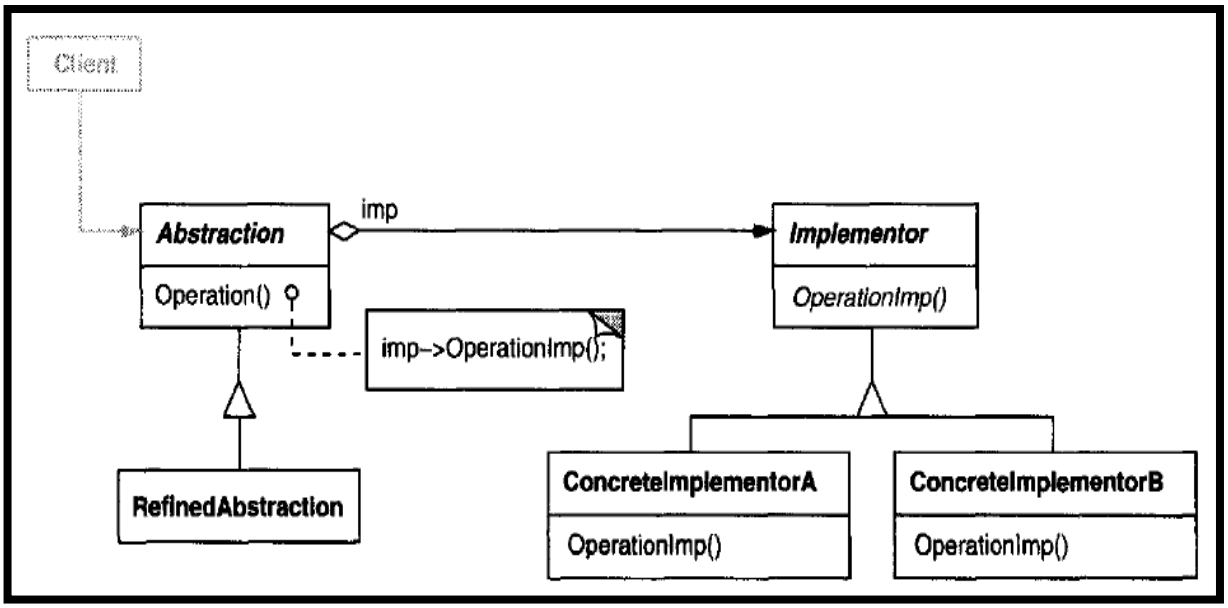
Use the Bridge pattern when

• You want to avoid a permanent binding between an abstraction and its implementation. This might be the case, for example, when the implementation must be selected or switched at run- time.

• Both the abstractions and their implementations should be extensible by sub classing. In this case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently.

• Changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.

Structure



Participants

- **Abstraction** (Window):
  - ✓ Defines the abstraction's interface.
  - ✓ Maintains a reference to an object of type Implementor.
- **RefinedAbstraction** (IconWindow):
  - ✓ Extends the interface defined by Abstraction.
- **Implementor** (WindowImp):
  - ✓ Defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different. Typically the Implementor interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.
- **ConcreteImplementor** (XWindowImp, PMWindowImp)
  - ✓ Implements the Implementor interface and defines its concrete implementation.

Collaborations

Abstraction forwards client requests to its Implementor object.

Consequences

The Bridge pattern has the following consequences:

- ✓ **Decoupling interface and implementation:** An implementation is not bound permanently to an interface. The implementation of an abstraction can be configured at run-time. It's even possible for an object to change its implementation at run-time.
- ✓ **Improved extensibility:** You can extend the Abstraction and Implementor hierarchies independently.

Sample Code

The following C+ + code implement the Window/Windowlmp example from the Motivation section. The Window class defines the window abstraction for client applications:

```
Class Window
{
   public:
   Window(View* contents); // requests handled by window
   virtual void DrawContents();
   virtual void Open();
   virtual void Close();
   virtual void Iconify();
   virtual void Deiconify();
// requests forwarded to implementation
   virtual void SetOrigin(const Points at);
   virtual void SetExtent(const Point& extent);
   virtual void Raise();
   virtual void Lower();
   virtual void DrawLine(const Point&, const Point&);
   virtual void DrawRect(const Point&, const Point&);
   virtual void DrawPolygon(const Point[], int n);
```

```
        Virtual void DrawText (const char*, const Point&);
Protected:
        WindowImp* GetWindowImp();
        View* GetView();
Private:
        WindowImp* _imp;
        View* _contents; // the window's contents
    };
```

Related Patterns

- ✓ An Abstract Factory can create and configure a particular Bridge.
- ✓ The Adapter pattern is geared toward making unrelated classes work together.

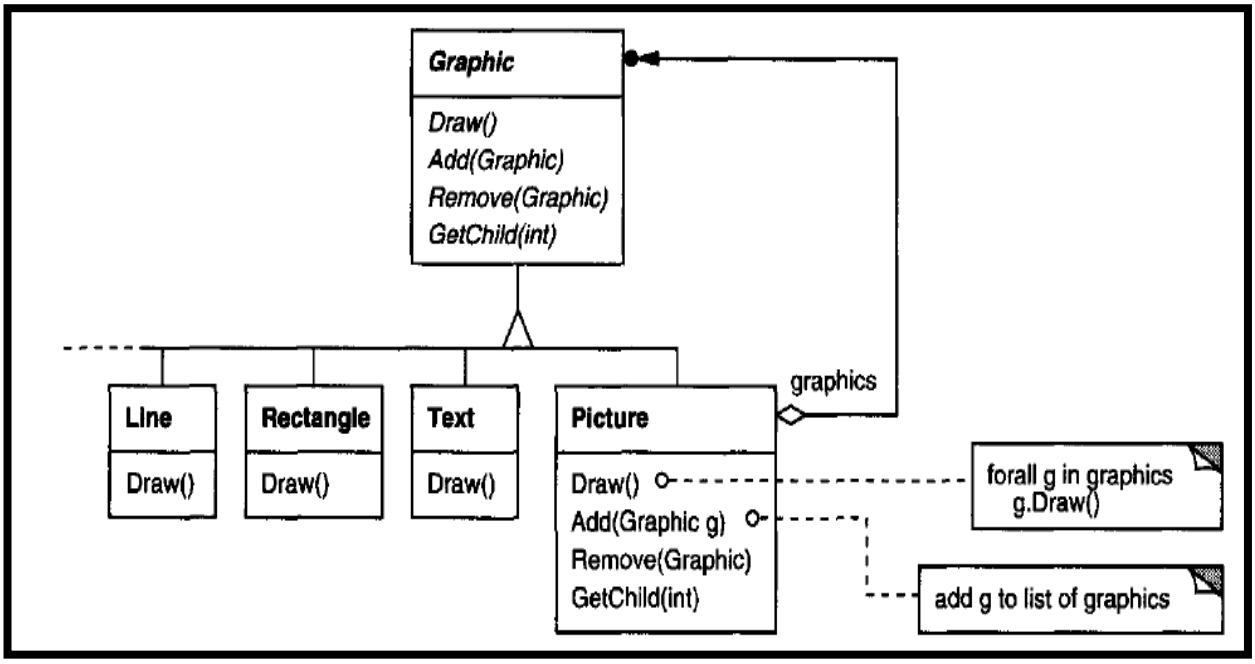
Composite Pattern

Intent

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

Motivation

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

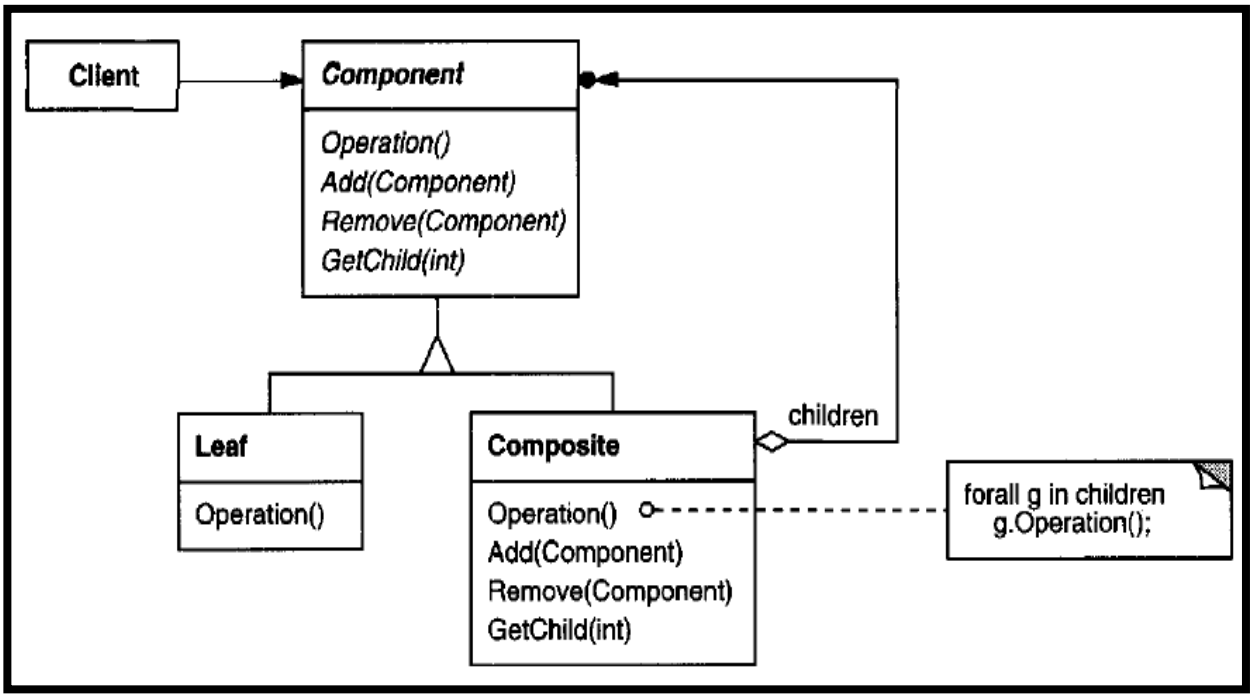


Applicability

Use the Composite pattern when

- 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.

Structure



Participants

- **Component (**Graphic**)**
- **Leaf (**Rectangle, Line**)**
- **Composite (**Picture**)**
- **Client**

Component (Graphic)

- ✓ 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.
- ✓ (Optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.

Leaf (Rectangle, Line, Text, etc,.)

- ✓ Represents leaf objects in the composition. A leaf has no children.
- ✓ Defines behavior for primitive objects in the composition.

Composite (Picture)

- ✓ 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.

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

The Composite pattern

- Defines class hierarchies consisting of primitive objects and composite objects. Primitive objects can be composed into more complex objects, which in turn can be composed, and soon recursively.
- Makes the client simple. Clients can treat composite structures and individual objects uniformly. Clients normally don't know (and shouldn't care) whether they're dealing with a leaf or a composite component.

Implementation

There are many issues to consider when implementing the Composite pattern:

- Explicit parent references. Maintaining references from child components to their parent can simplify the traversal and management of a composite structure.
- Sharing components. It's often useful to share components, for example, to reduce storage requirements.

Sample Code

Equipment class defines an interface for all equipment in the part-whole hierarchy.

```
Class Equipment
    {
        Public:
        Virtual "Equipment ();
        Const char* Name () {return _name; }
        Virtual Watt Power ();
        Virtual Currency NetPrice ();
        Virtual Currency DiscountPrice ();
        Virtual void Add (Equipment*);
        Virtual void Remove (Equipment*);
        Virtual Iterator<Equipment*>* CreateIterator ();
Protected:
        Equipment (const char*);
Private:
        Const char* _name;
    };
```

Related Patterns

- Often the component-parent link is used for a Chain of Responsibility
- Flyweight lets you share components, but they can no longer refer to their parents
- Iterator can be used to traverse composites.

Decorator Pattern

Intent

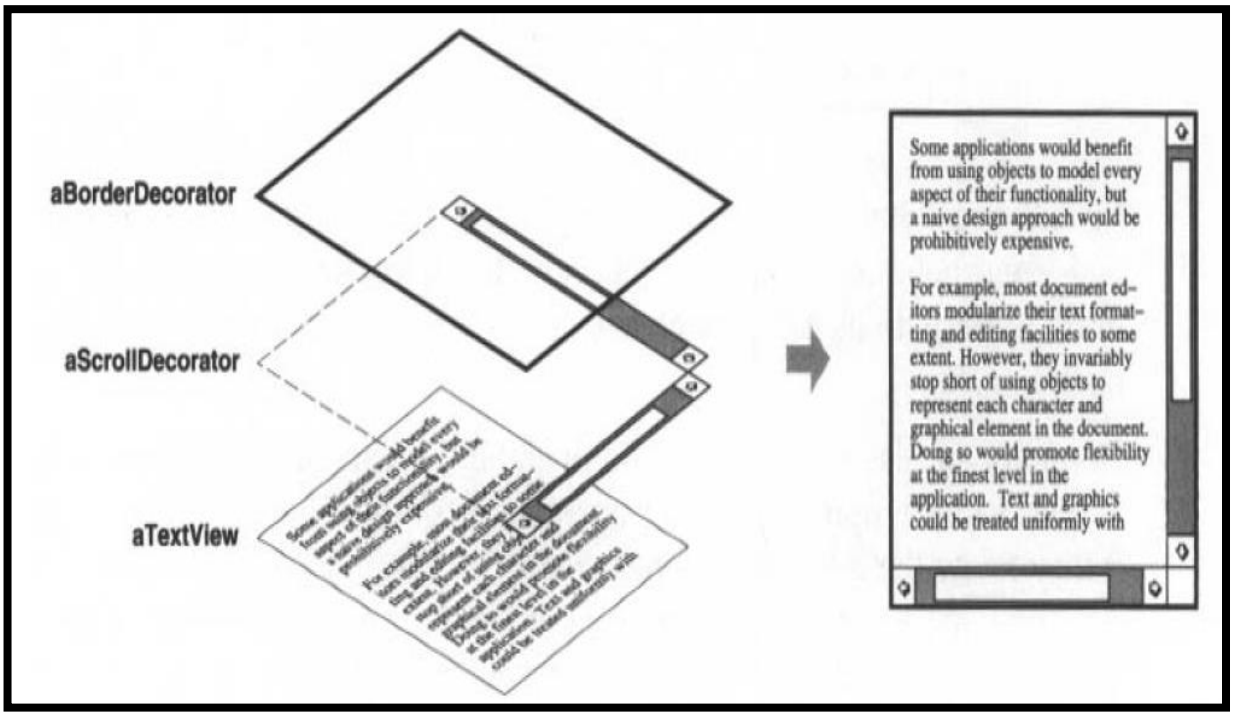
Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.

Also Known As

Wrapper

Motivation

Sometimes we want to add responsibilities to individual objects, not to an entire class. A graphic al user interface toolkit, for example, should let you add properties like borders or behaviors like scrolling to any user interface component.

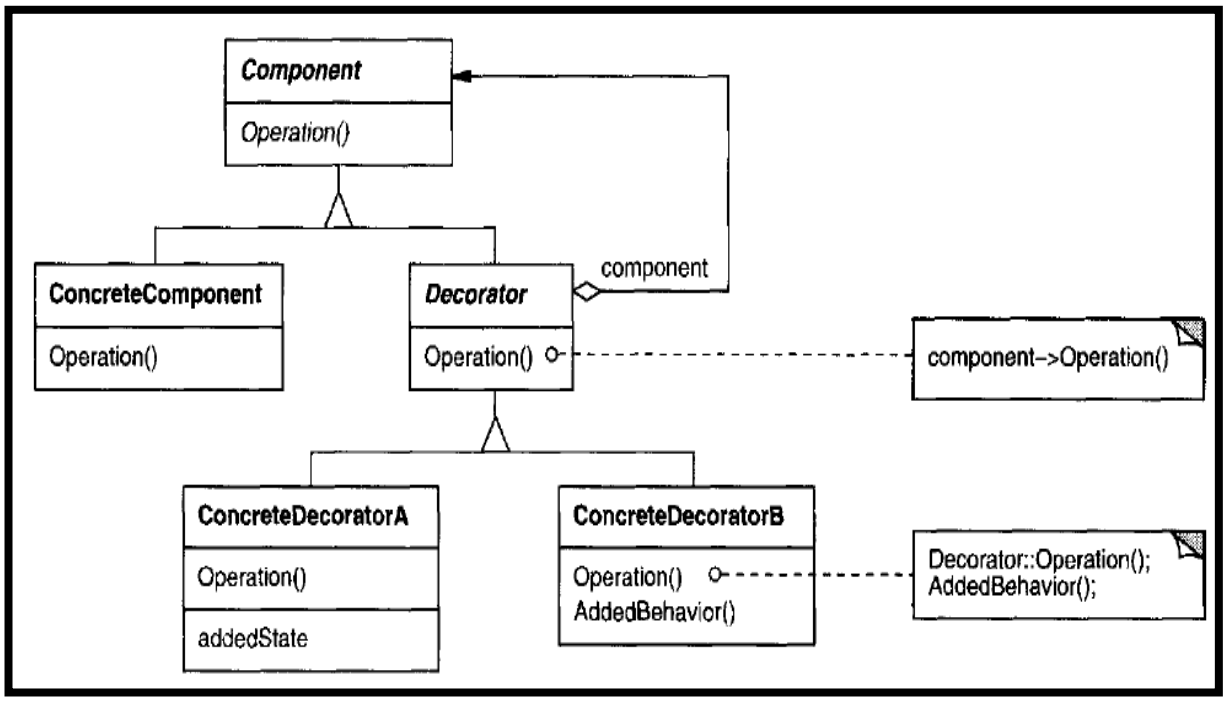


Applicability

Use Decorator

- To add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
- For responsibilities that can be withdrawn.

Structure



Participants

• **Component (**Visual Component**)**

- ✓ Defines the interface for objects that can have responsibilities added to them dynamically.

• **Concrete Component (**TextView**)**

- ✓ 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.

Collaborations

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

Consequences

The Decorator pattern has at least two key benefits and two liabilities:

- ✓ More flexibility than static inheritance.
- ✓ Avoids feature-laden classes high up in the hierarchy.

Implementation

Several issues should be considered when applying the Decorator pattern

- ✓ **Interface conformance** A decorator object's interface must conform to the interface of the component it decorates.
- ✓ Omitting the abstract Decorator class.

Sample Code

```
Class VisualComponent
      {
         Public:
         VisualComponent ();
         Virtual void Draw ();
         Virtual void Resize ();
      };
Class Decorator: public VisualComponent
     {
       Public:
           Decorator (VisualComponent*);
           Virtual void Draw ();
           Virtual void Resize ();
     Private:
           VisualComponent* _component;
      };
```

Related Patterns

- **Adapter**: A decorator is different from an adapter in that a decorator only changes an object's responsibilities, not its interface; an adapter will give an object a completely new interface.
- **Composite**: A decorator can be viewed as a degenerate composite with only one component.
- **Strategy**: A decorator lets you change the skin of an object; a strategy lets you change the guts. These are two alternative ways of changing an object.

Facade Pattern

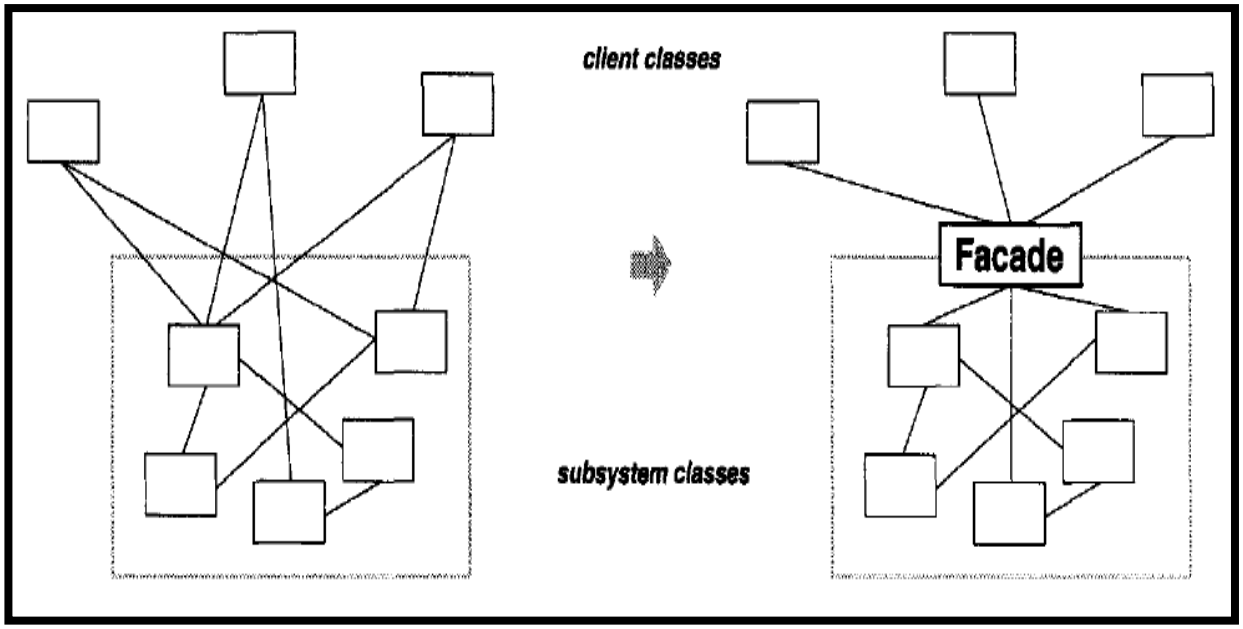
Intent

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.

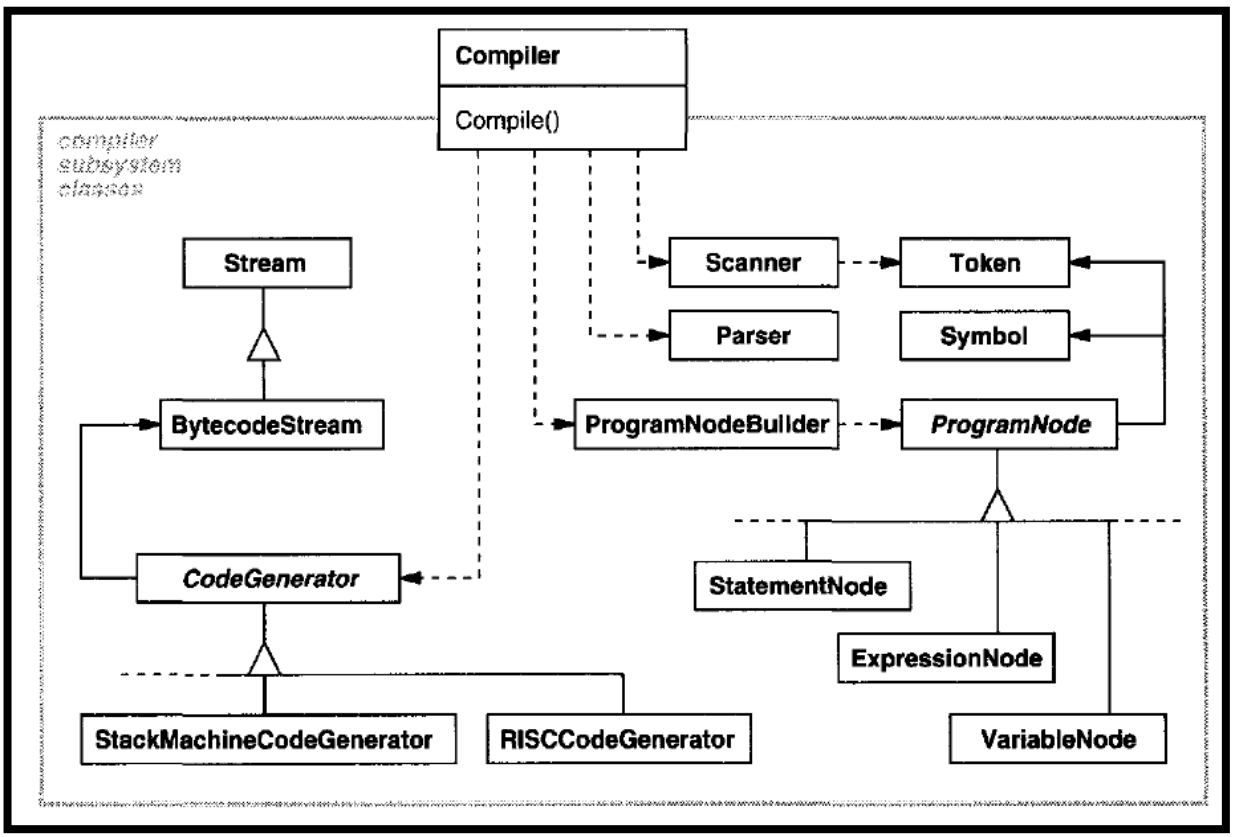
Motivation

Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems.

One way to achieve this goal is to introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem.



Consider for example a programming environment that gives applications access to its compiler subsystem. This subsystem contains classes such as Scanner, Parser, ProgramNode, BytecodeStream, and ProgramNodeBuilder that implement the compiler. Some specialized applications might need to access these classes directly.

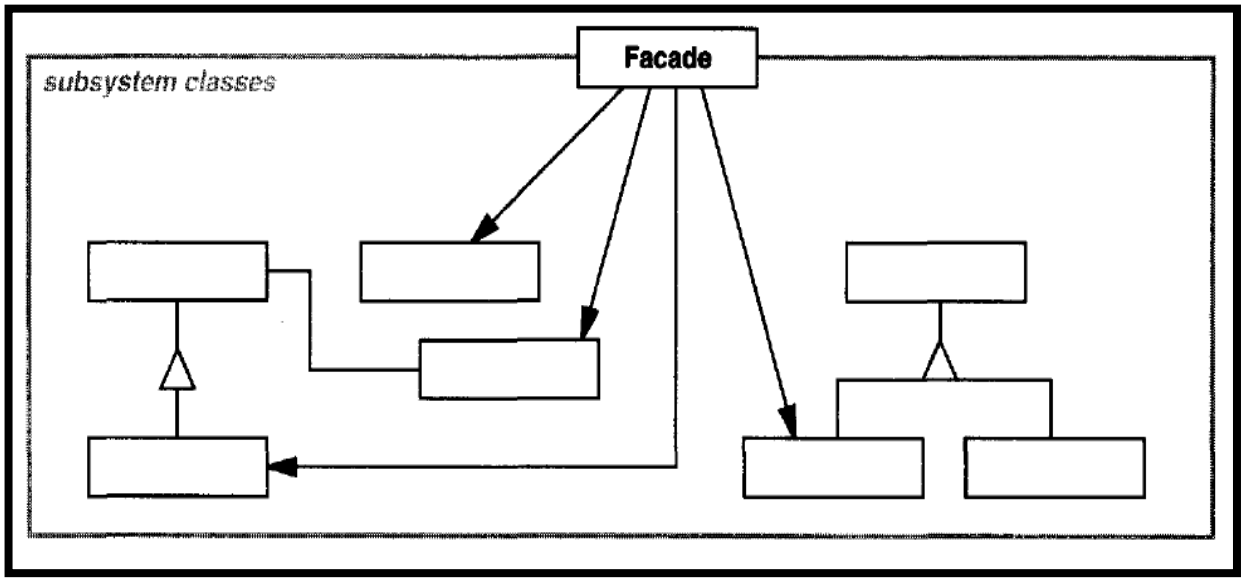


Applicability

Use the Façade pattern when

- To provide simple interface to a complex package, which is useful for most clients?
- To reduce the dependencies between the client and the package, or dependencies between various packages.

Structure



Participants

- **Facade (**Compiler**)**
  - ✓ Knows which subsystem classes are responsible for a request.
  - ✓ Delegate's client requests to appropriate subsystem objects.
- **subsystem classes** (Scanner, Parser, ProgramNode, etc.)
  - ✓ Implement subsystem functionality.
  - ✓ Handle work assigned by the Facade object.
  - ✓ Have no knowledge of the facade; that is, they keep no references to it.

Collaborations

- Clients communicate with the subsystem by sending requests to Facade, which forwards them to the appropriate subsystem object(s).
- Although the subsystem objects perform the actual work, the facade may have to do work of its own to translate its interface to subsystem interfaces.
- Clients that use the facade don't have to access its subsystem objects directly.

Consequences

- ✓ Shields a client from the low-level classes of a subsystem.
- ✓ Simplifies the use of a subsystem by providing higher-level methods.
- ✓ Enables lower-level classes to be restructured without changes to clients.

Note: The repeated use of Facade patterns yields a layered system.

Related Patterns

Abstract Factory: it can be used with Facade to provide an interface for creating subsystem objects in a subsystem-independent way. Abstract Factory can also be used as an alternative to Facade to hide platform-specific classes.

Mediator: it is similar to Facade in that it abstracts functionality of existing classes.

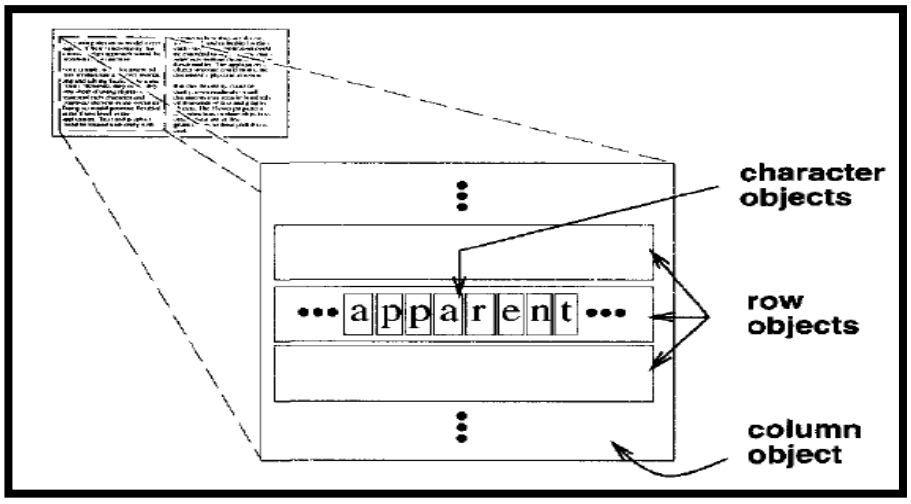
Flyweight Pattern

Intent

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

Motivation

Some applications could benefit from using objects throughout their design, but a naive implementation would be prohibitively expensive. For example, most document editor implementations have text formatting and editing facilities that are modularized to some extent. Object-oriented document editors typically use objects to represent embedded elements like tables and figures.



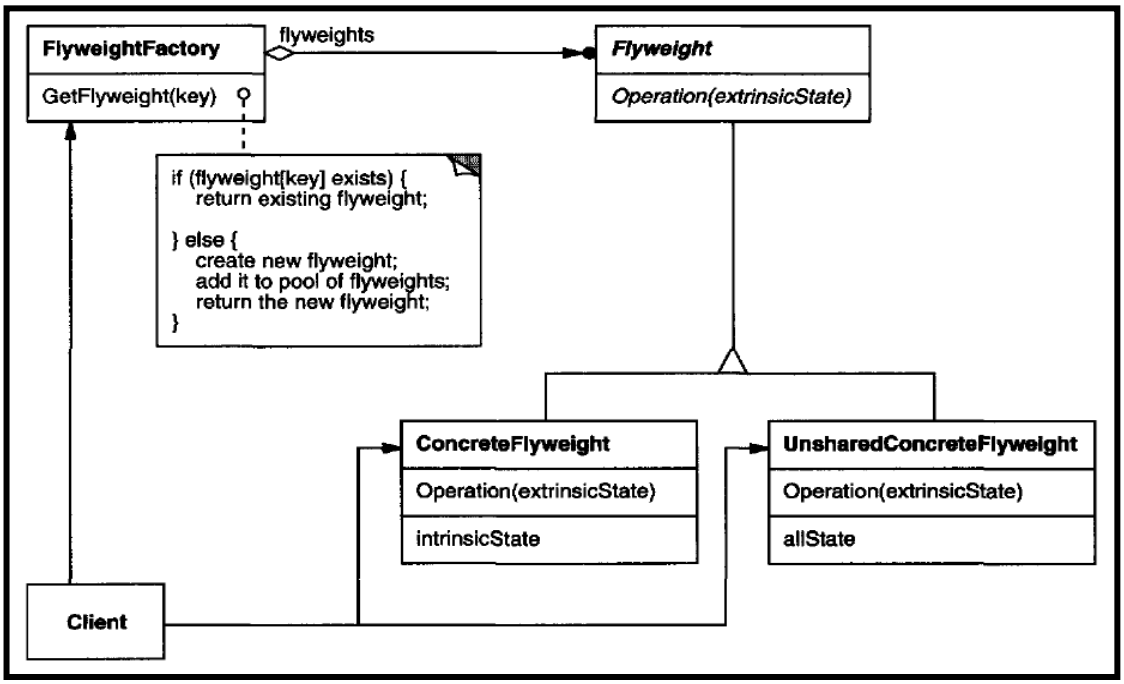
A **flyweight** is a shared object that can be used in multiple contexts simultaneously. The flyweight acts as an independent object in each context—it's indistinguishable from an instance of the object that's not shared. Flyweights cannot make assumptions about the context in which they operate. The key concept here is the distinction between intrinsic and extrinsic state. Intrinsic state is stored in the flyweight; it consists of information that's independent of the flyweight's context, thereby making it sharable. Extrinsic state depends on and varies with the flyweight's context and therefore can't be shared. Client objects are responsible for passing extrinsic state to the flyweight when it needs it.

Applicability

The Flyweight pattern's effectiveness depends heavily on how and where it's used. Apply the Flyweight pattern when all of the following are true:

- An application uses a large number of objects.
- Storage costs are high because of the sheer quantity of objects
- Most object state can be made extrinsic.
- Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.

Structure



Participants

Flyweight (Glyph**)**

- ✓ Declares an interface through which flyweights can receive and act on extrinsic state.

ConcreteFlyweight (Character**)**

- ✓ Implements the Flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be sharable.

UnsharedConcreteFlyweight (Row ,Column**)**

- ✓ Not all Flyweight subclasses need to be shared. The Flyweight interface enables sharing; it doesn't enforce it.

FlyweightFactory

- ✓ Creates and manages flyweight objects.
- ✓ Ensures that flyweights are shared properly. When a client requests a flyweight, the FlyweightFactory object supplies an existing instance or creates one, if none exists.

Client

- ✓ Maintains a reference to flyweight(s).
- ✓ Computes or stores the extrinsic state of flyweight(s).

Collaborations

- State that a flyweight needs to function must be characterized as either intrinsic or extrinsic. Intrinsic state is stored in the ConcreteFlyweight object; extrinsic state is stored or computed by Client objects. Clients pass this state to the flyweight when they invoke its operations.
- Clients should not instantiate ConcreteFlyweights directly. Clients must obtain ConcreteFlyweight objects exclusively from the FlyweightFactory object to ensure they are shared properly.

Consequences

- The reduction in the total number f instances t hat comes from sharing
- The amount of intrinsic state per object
- Whether extrinsic state is computed or stored

Implementation

Consider the following issues when implementing the Flyweight pattern:

- ❖ Removing extrinsic state
- ❖ Managing shared objects

Sample Code

Returning to our document formatter example, we can define a Glyph base class for flyweight graphical objects. Logically, glyphs are Composites that have graphic al attributes and ca n draw themselves. Here we focus on just the font attribute, but the same approach can be used for any other graphical attributes a glyph might have.

```
class Glyph {
public:
    virtual ~Glyph();

    virtual void Draw(Window*, GlyphContext&);


    virtual void SetFont(Font*, GlyphContext&);
    virtual Font* GetFont(GlyphContext&);


    virtual void First(GlyphContext&);
    virtual void Next(GlyphContext&);
    virtual bool IsDone(GlyphContext&);
    virtual Glyph* Current(GlyphContext&);


    virtual void Insert(Glyph*, GlyphContext&);
    virtual void Remove(GlyphContext&);
protected:
    Glyph();
};
```

Related Patterns

- The Flyweight pattern is often combined with the Composite pattern to implement a logically hierarchical structure in terms of a directed-acyclic graph with shared leaf nodes.
- It's often best to implement State and Strategy objects as flyweights

Proxy Pattern

Intent

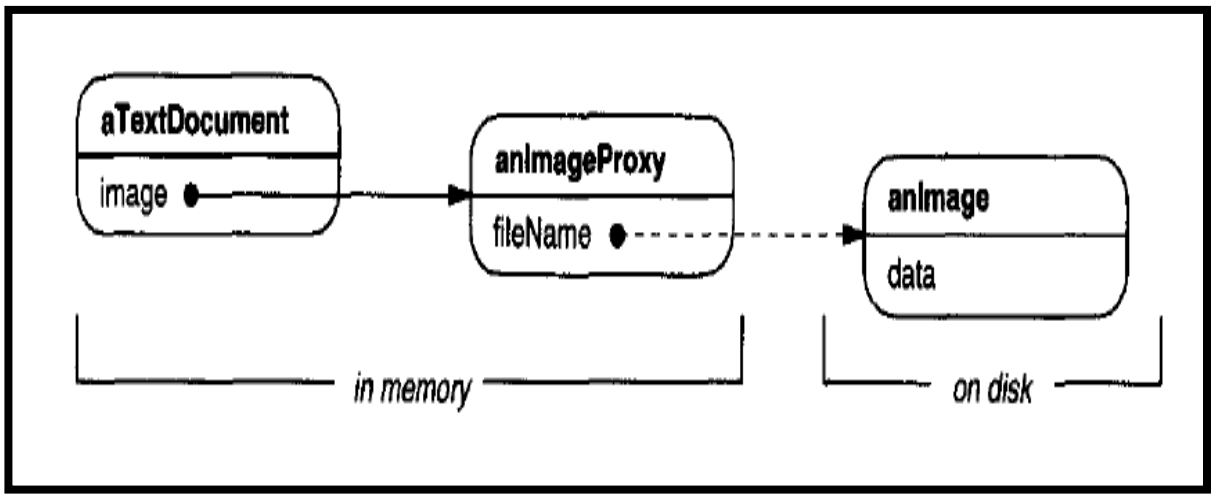
Provide a surrogate or placeholder for another object to control access to it.

Also Known As

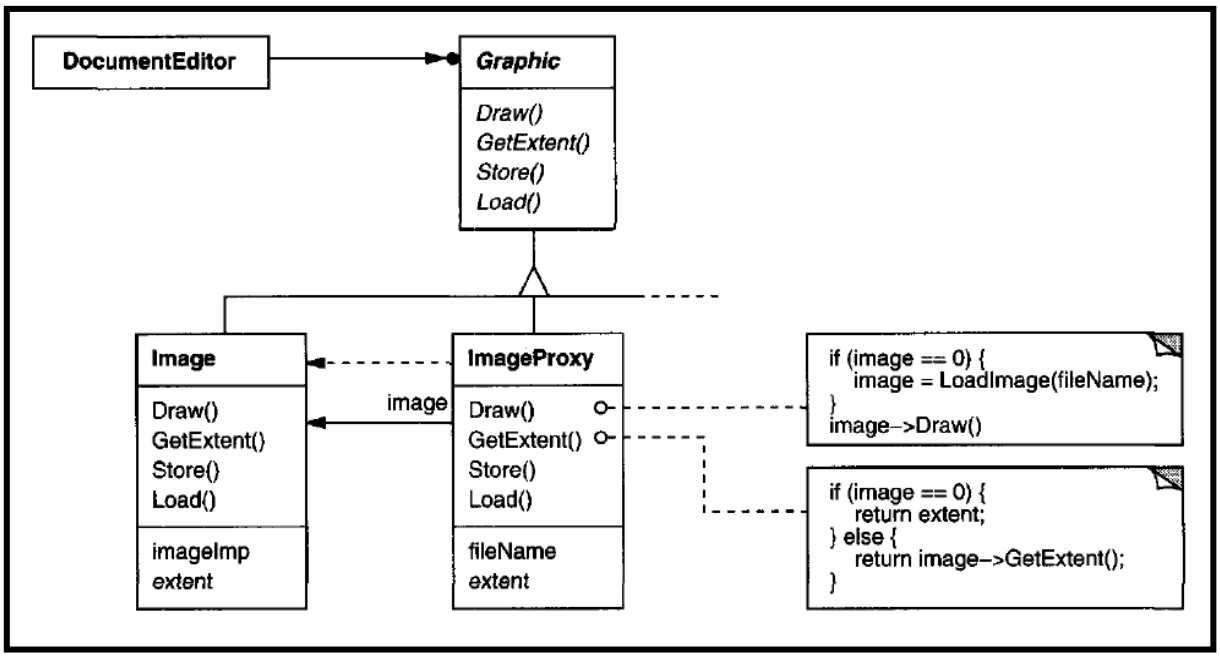
Surrogate

Motivation

One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it. Consider a document editor that can embed graphical objects in a document. Some graphical objects, like large raster images, can be expensive to create. But opening a document should be fast, so we should avoid creating all the expensive objects at once when the document is opened. This isn't necessary anyway, because not all of these objects will be visible in the document at the same time.



The following class diagram illustrates this example in more detail.

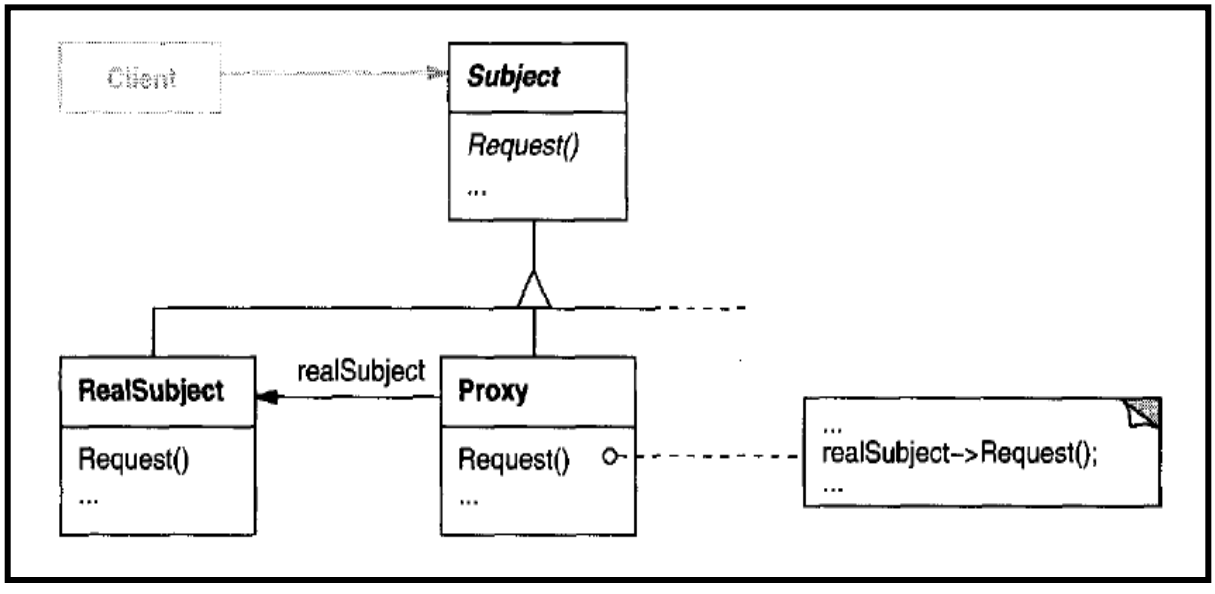


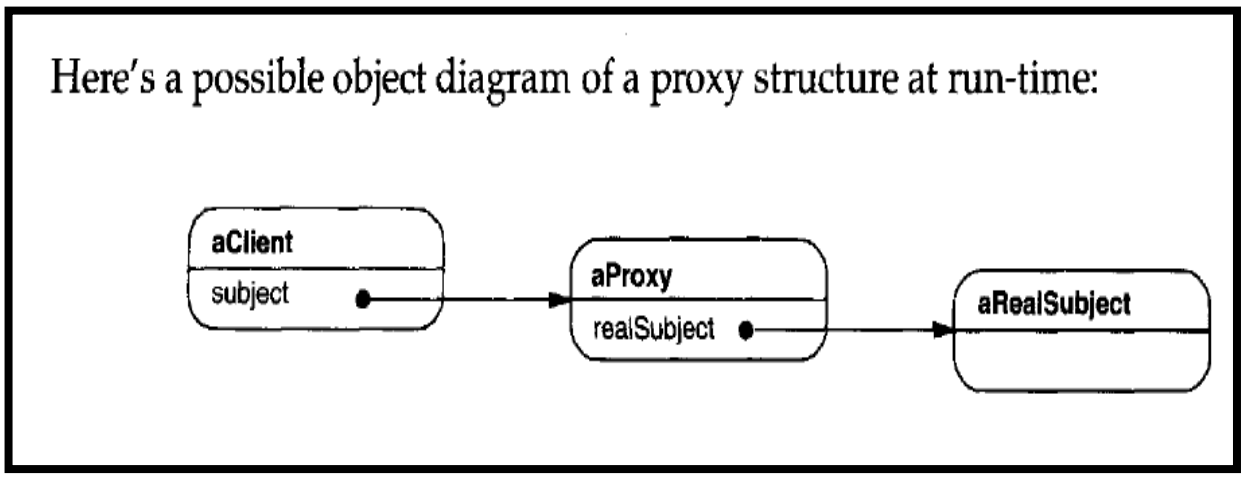
Applicability

Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. Here are several common situations in which the Proxy pattern is applicable:

- A remote proxy provides a local representative for an object in a different address space.
- A virtual proxy creates expensive objects on demand. The Image Proxy described in the Motivation is an example of such a proxy.
- A protection proxy controls access to the original object. Protection proxies are useful when objects should have different access rights.

Structure





Participants

Proxy (ImageProxy**)**

- ✓ Maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
- ✓ Provides an interface identical to Subject's so that a proxy can be substituted for the real subject.
- ✓ Controls access to the real subject and may be responsible for creating and deleting it.

Subject (Graphic**)**

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

RealSubject (Image)

- ✓ Defines the real object that the proxy represents.

Collaborations

- Proxy forwards requests to RealSubject when appropriate, depending on the kind of proxy

Consequences

The Proxy pattern introduces a level of indirection when accessing an object. The additional indirection has many uses, depending on the kind of proxy:

- A remote proxy can hide the fact that an object resides in a different address space
- A virtual proxy can perform optimizations such as creating an object on demand
- Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed.

Sample Code

The following code implements two kinds of proxy: the virtual proxy described in the Motivation section, and a proxy implemented with does Not Understand.

1. *A virtual proxy.* The `Graphic` class defines the interface for graphical objects:

```
class Graphic {
public:
    virtual ~Graphic();

    virtual void Draw(const Point& at) = 0;
    virtual void HandleMouse(Event& event) = 0;

    virtual const Point& GetExtent() = 0;

    virtual void Load(istream& from) = 0;
    virtual void Save(ostream& to) = 0;
protected:
    Graphic();
};
```

Related Patterns

- **Adapter:** An adapter provides a different interface to the object it adapts.
- **Decorator:** Although decorators can have similar implementations as proxies, decorators have a different purpose. A decorator adds one or more responsibilities to an object, whereas a proxy controls access to an object.

Frequently Asked Question

1. Write a short note on Adapter design pattern?
2. Briefly discuss about Bridge design pattern?
3. Explain the working of Composite design pattern?
4. Explain in detail about Decorator design pattern?
5. Describe the working of Façade design pattern?
6. Write about Flyweight design pattern?
7. Briefly mention about the working of PROXY design pattern?