Structural Patterns

Structural patterns are concerned with how classes and objects are composed to form larger structures. Structural class patterns use inheritance to compose interfaces or implementations. As a simple example, consider how multiple inheritances mix two or more classes into one. The result is a class that combines the properties of its parent classes.

ADAPTER

Class, Object Structural

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.

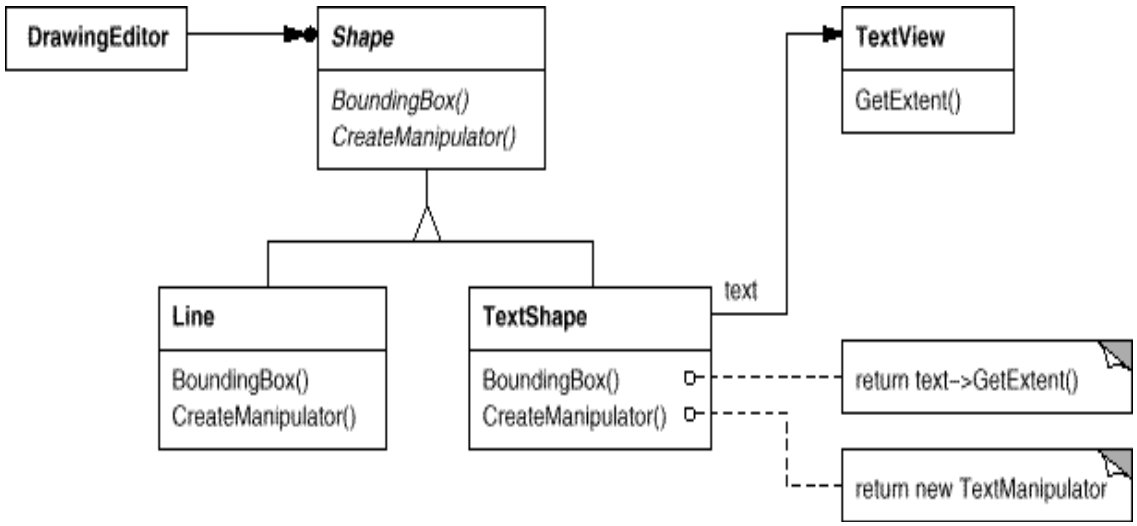
Also Known As

Wrapper

Motivation

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.

This diagram (below) illustrates the object adapter case. It shows how Bounding Box requests, declared in class Shape, are converted to GetExtent requests defined in TextView.Since TextShape adapts TextView to the Shape interface, the drawing editor canreuse the otherwise incompatible TextView class.



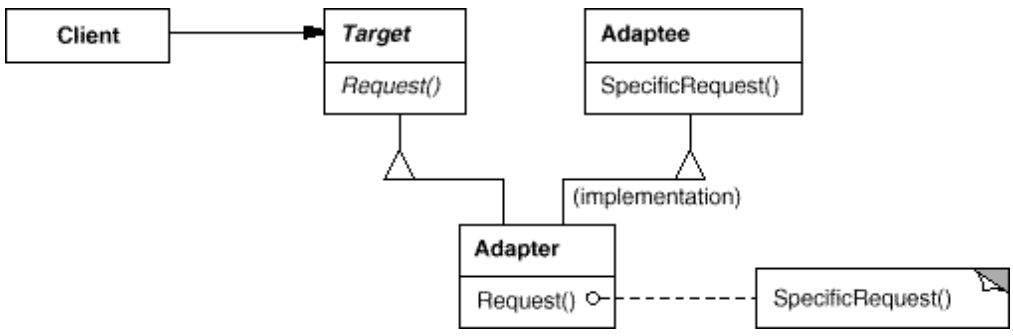
Applicability

We use the Adapter pattern when

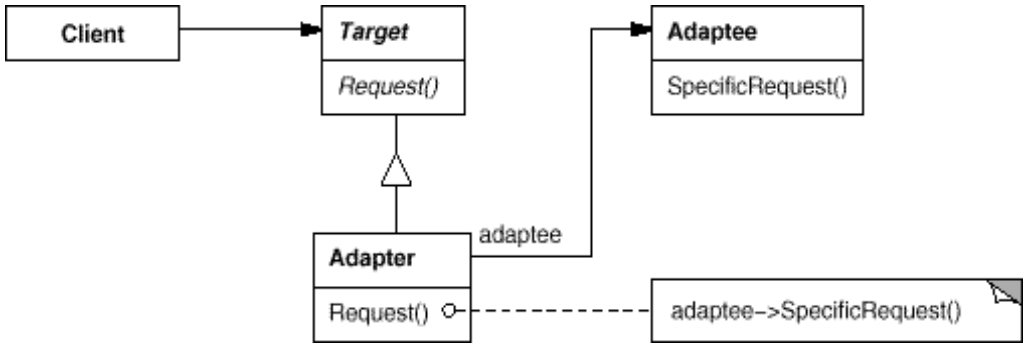
- We want to use an existing class, and its interface does not match the one you need.
- We want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
- (**object adapter only**) We need to use several existing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

Structure

A class adapter uses multiple inheritance 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.

- **Adaptee** (TextView)

  -defines an existing interface that needs adapting.

- **Adapter** (TextShape)

  -adapts the interface of Adaptee to the Target interface.

Collaborations

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

Consequences

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

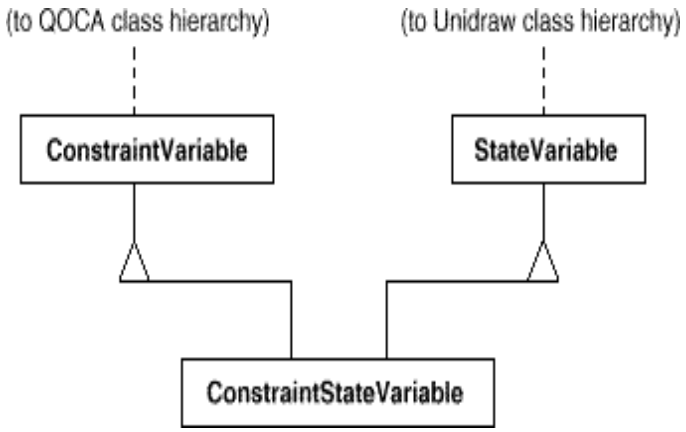
- Adapts Adaptee to Target by committing to a concrete Adapter class. As a consequence, a class adapter won't work when we want to adapt a class and all its subclasses.
- let’s Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
- Introduces only one object, and no additional pointer indirection is needed to get to the adaptee.

An object adapter

- Let's 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.

Here are other issues to consider when using the Adapter pattern:

1. How much adapting does Adapter do?
2. Pluggable adapters
3. Using two-way adapters to provide transparency.



Above diagram shows a two-way class adapter Constraint State Variable, a subclass of both State Variable and Constraint Variable that adapts the two interfaces to each other. Multiple inheritance is a viable solution in this case because the interfaces of the adapted classes are substantially different.

Implementation

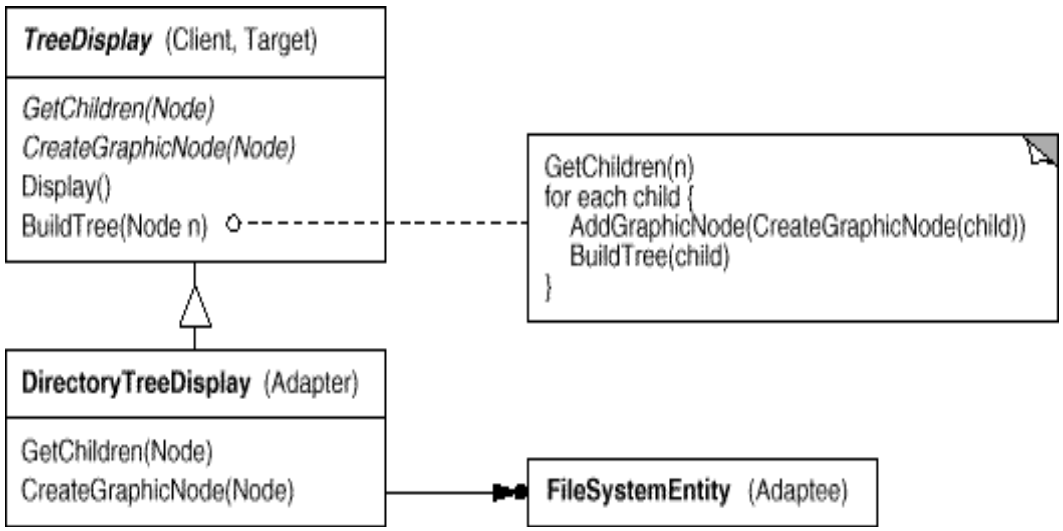
Although the implementation of Adapter is usually straightforward, here are some issues to keep in mind:

1. Implementing class adapters in C++.
2. Pluggable adapters.

The first step, which is common to all three of the implementations discussed here, is to find a "narrow" interface for Adaptee, that is, the smallest subset of operations that lets us do the adaptation.

The narrow interface leads to three implementation approaches:

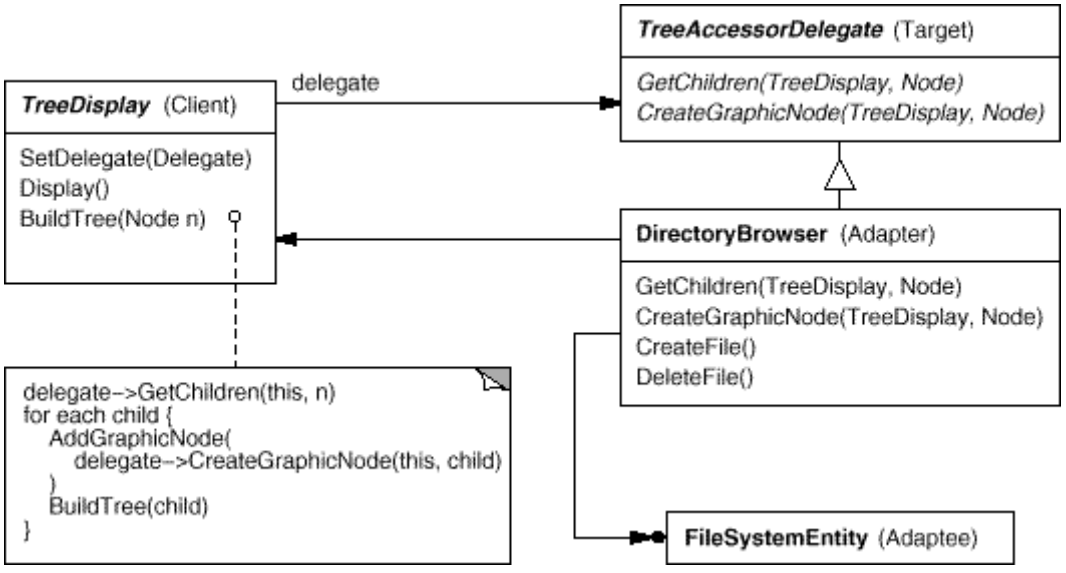
1. **Using abstract operations**.



Above diagram shows an example of Directory Tree Display subclass will implement these operations by accessing the directory structure.

2. Using delegate objects.

In this approach, TreeDisplay forwards requests for accessing the hierarchical structure to a delegateobject. TreeDisplay can use a different adaptation strategy by substituting a different delegate.



For example, suppose there exists a Directory Browser that uses aTreeDisplay. DirectoryBrowser might make a good delegate for adapting TreeDisplay to the hierarchical directory structure.

3. **Parameterized adapters**. The usual way to support pluggable adapters in Smalltalk is to parameterize an adapter with one or more blocks.The block construct supports adaptation without subclassing.

For example, to create TreeDisplay on a directory hierarchy, we write

```
directoryDisplay :=
    (TreeDisplay on: treeRoot)
        getChildrenBlock:
            [:node | node getSubdirectories]
        createGraphicNodeBlock:
            [:node | node createGraphicNode].
```

Sample Code

We'll give a brief sketch of the implementation of class and object adapters forthe 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;
};
classTextView {
public:
    TextView();
```

```
    voidGetOrigin(Coord& x, Coord& y) const;
    voidGetExtent(Coord& width, Coord& height) const;
    virtual bool IsEmpty() const;
};
```

Shape assumes a bounding box defined by its opposing corners. In contrast, TextViewis defined by an origin, height, and width. Shape also defines a Create Manipulator operation for creating a Manipulator object, which knows how to animate a shape when the user manipulates it. TextView has no equivalent operation. The classTextShape is an adapter between these different interfaces.

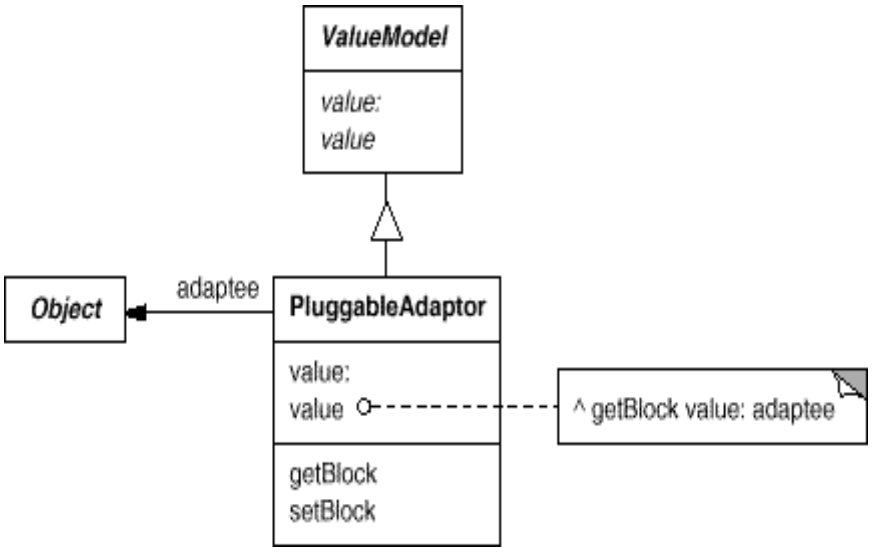
Known Uses

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

[WGM88]. ET++Draw reuses the ET++ classes for text editing by using a TextShape

adapter class.

Pluggable adapters are common in ObjectWorks\Smalltalk [Par90]. Standard Smalltalk defines a Value Model class for views that display a single value.



ValueModel (above diagram) defines a value, value: interface for accessing the value.

Related Patterns

Bridge has a structure similar to an object adapter, but Bridge has a different intent: It is meant to

separate an interface from its implementation so that they can be varied easily and independently. An adapter is meant to change the interface of an existing object. Decorator enhances another object without changing its interface. Proxy defines a representative or surrogate for another object and does not change its interface.

PROXY

Object Structural

Intent

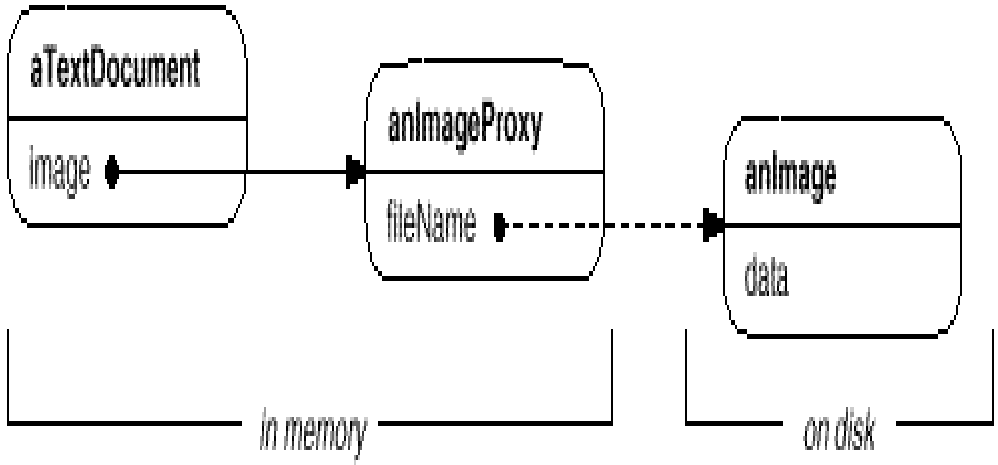
Provide a surrogate or placeholder for another object to control access to it.

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

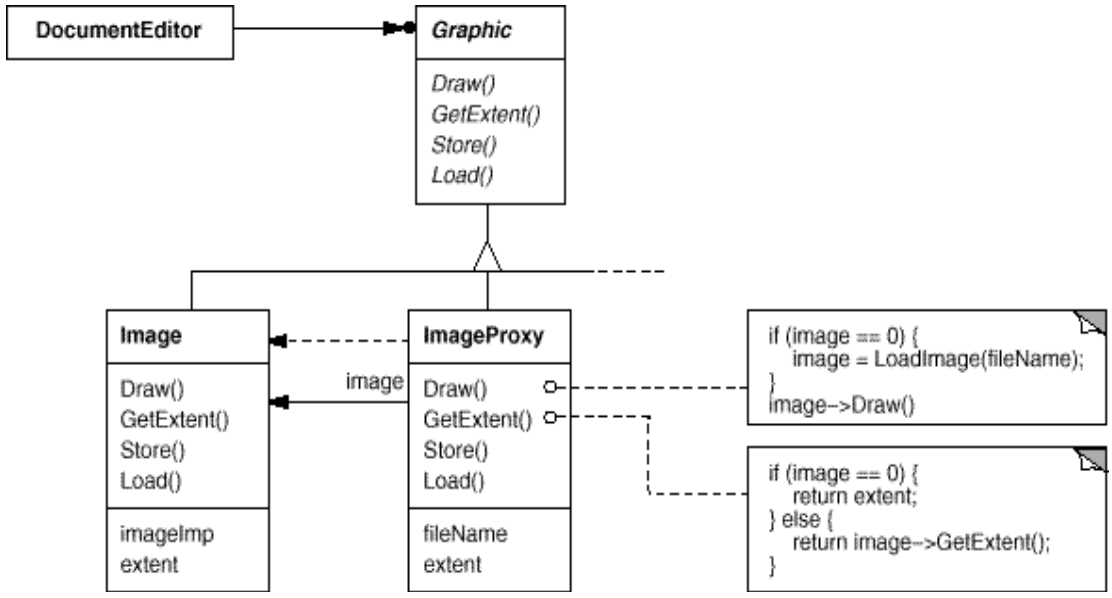


expensive to create.

The image proxy creates the real image only when the document editor asks it to display itself by invoking its Draw operation. The proxy forwards subsequent requests directly to the image. It must therefore keep a reference to the image after creating it.

Let's assume that images are stored in separate files. In this case we can use the file name as the reference to the real object. The proxy also stores its extent,that is, its width and height. The extent lets the proxy respond to requests for its size from the formatter without actually

instantiating the image.



Above diagram shows the DocumentEditorclass diagram 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:

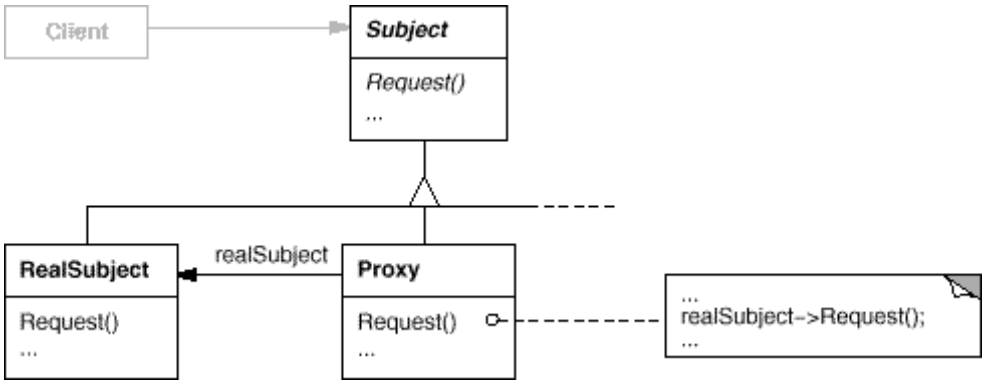
1. **A remote proxy** provides a local representative for an object in a different address space. NEXTSTEP [Add94] uses the class NXProxy for this purpose. Coplien [Cop92] calls this kind of proxy an "Ambassador."
2. A **virtual proxy** creates expensive objects on demand. The Image Proxy described in the Motivation is an example of such a proxy.
3. A **protection proxy** controls access to the original object. Protection proxies are useful when objects should have different access rights. For example, KernelProxies in the Choices operating system [CIRM93] provide protected access to operating system objects.
4. A smart reference is a replacement for a bare pointer that performs additional actions when an object is accessed. Typical uses include

   -counting the number of references to the real object so that it can be freed automatically when there are no more references (also called **smart pointers** [Ede92]).

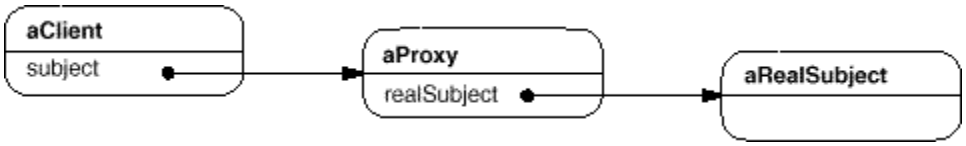
   -loading a persistent object into memory when it's first referenced.

-checking that the real object is locked before it's accessed to ensure that no other object can change it.

Structure



Here's a possible object diagram of a proxy structure at run-time:



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 interfacesare the same.

  -provides an interface identical to Subject's so that a proxy canby substituted for the real subject.

  -controls access to the real subject and may be responsible for creating and deleting it.

-other responsibilities depend on the kind of proxy:

- Remote proxies are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.

- Virtual proxies may cache additional information about the real subject so that they can postpone accessing it. For example, the Image Proxy from the Motivation caches the realimage's extent.
- protection proxies check that the caller has the access permissions required to perform a request.

- **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:

1. A remote proxy can hide the fact that an object resides in a different address space.

2. A virtual proxy can perform optimizations such as creating an object on demand.

3. Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed.

Implementation

The Proxy pattern can exploit the following language features:

1. **Overloading the member access operator in C++.**

   The following example illustrates how to use this technique to implement

   a virtual proxy called ImagePtr.

   class Image;

```
extern Image* LoadAnImageFile(const char*);
    // external function
classImagePtr {
public:
    ImagePtr(const char* imageFile);
    virtual ~ImagePtr();
    virtual Image* operator->();
    virtual Image& operator*();
private:
    Image* LoadImage();
private:
    Image* _image;
    const char* _imageFile;
};
ImagePtr::ImagePtr (const char* theImageFile) {
    _imageFile = theImageFile;
    _image = 0;
}
Image* ImagePtr::LoadImage () {
    if (_image == 0) {
            _image = LoadAnImageFile(_imageFile);
}
return _image;
}
```

The overloaded -> and * operators use LoadImage to return _image to callers(loading it if necessary).

```
Image* ImagePtr::operator-> () {

returnLoadImage();

}

Image&ImagePtr::operator* () {

return *LoadImage();

}
```

2. **Using doesNotUnderstand in Smalltalk.** Smalltalk provides a hook that you can use to support automatic forwarding of requests. Smalltalk calls *doesNotUnderstand: aMessage* when a client sends a message to a receiver that has no corresponding method. The main disadvantage of doesNot Understand: is that most Smalltalk systems have a few special messages that are handled directly by the virtual machine, and these do not cause the usual method look-up.
3. **Proxy doesn't always have to know the type of real subject**. If a Proxy class can deal with its subject solely through an abstract interface, then there's no need to make a Proxy class for each RealSubject class; the proxy can deal with all RealSubject classes uniformly.

Another implementation issue involves how to refer to the subject before it's instantiated.

Sample Code

The following code implements two kinds of proxy: the virtual proxy described in the Motivation section, and a proxy implemented with doesNot 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;
        virtualconst Point&GetExtent() = 0;
        virtual void Load(istream& from) = 0;
        virtual void Save(ostream& to) = 0;
    protected:
                Graphic();
    };
```

2. **Proxies that use doesNotUnderstand**. You can make generic proxies inSmalltalk by defining classes whose superclass is nil8 and defining the doesNot Understand: method to handle messages.

```
doesNotUnderstand: aMessage
^ selfrealSubject
perform: aMessage selector
withArguments: aMessage arguments.
```

Known Uses

The virtual proxy example in the Motivation section is from the ET++ text buildingblock classes.NEXTSTEP [Add94] uses proxies (instances of class NXProxy) as localrepresentatives for objects that may be distributed. A server creates proxiesfor remote objects when clients request them. On receiving a message, the proxyencodes it along with its arguments and then forwards the encoded message to theremote subject.

Related Patterns

Adapter: An adapter provides a different interface to the object it adapts.In contrast, a proxy provides the same interface as its subject.

Decorator: Although decorators can have similar implementations as proxies, decorators have a different purpose.

Proxies vary in the degree to which they are implemented like a decorator. Aprotection proxy might be implemented exactly like a decorator.

Bridge

Intent

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

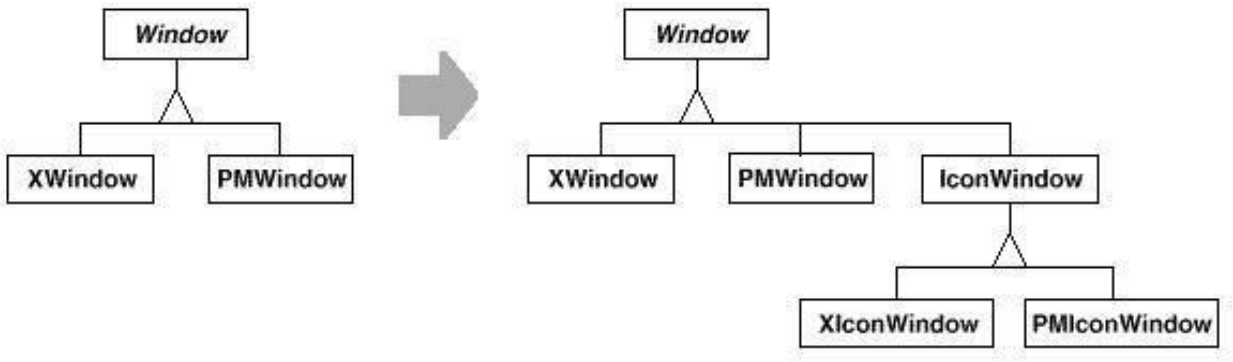
Also Known As

Handle/Body

Motivation

Consider the implementation of a portable Window abstraction in a user interface toolkit. This abstraction should enable us to write applications that work on both the X Window System and IBM's Presentation Manager (PM), for example. Using inheritance, we could define an abstract class Window and subclasses XWindow and PMWindow that implement the Window interface for the different platforms. **But this approach has two drawbacks:**

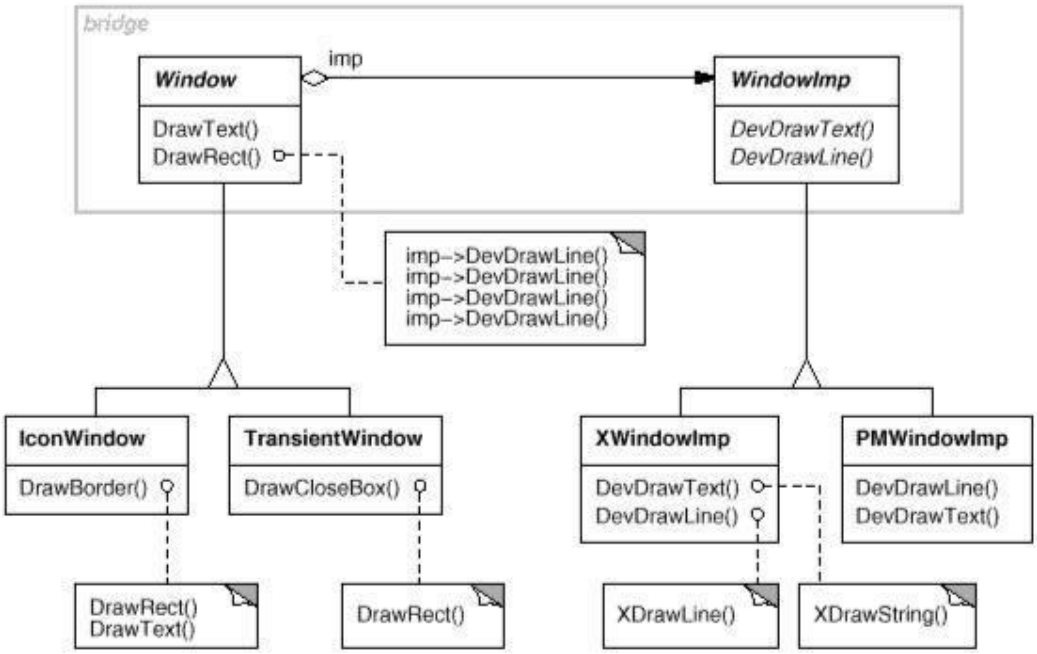
1. It's inconvenient to extend the Window abstraction to cover different kinds of windows or new platforms. Imagine an IconWindow subclass of Window that specializes the Window abstraction for icons. To support IconWindows for both platforms, we have to implement two new classes, XIconWindow and PMIconWindow.



2. It makes client code platform-dependent. Whenever a client creates a window, it instantiates a concrete class that has a specific implementation. For example, creating an XWindow object binds the Window abstraction to the X Window implementation, which makes the client code dependent on the X Window implementation. This, in turn, makes it harder to port the client code to other platforms.

Clients should be able to create a window without committing to a concrete implementation. Only the window implementation should depend on the platform on which the application runs. Therefore client code should instantiate windows without mentioning specific platforms.

The Bridge pattern addresses these problems by putting the Window abstraction and its implementation in separate class hierarchies. There is one class hierarchy for window interfaces (Window, IconWindow, TransientWindow) and a separate hierarchy for platform-specific window implementations, with WindowImp as its root.



The XWindowImp subclass, for example, provides an implementation based on the X Window System.

All operations on Window subclasses are implemented in terms of abstract operations from the WindowImp interface. This decouples the window abstractions from the various

platform-specific implementations. We refer to the relationship between Window and WindowImp as a bridge, because it bridges the abstraction and its implementation, letting them vary independently.

Applicability

Use the Bridge pattern when

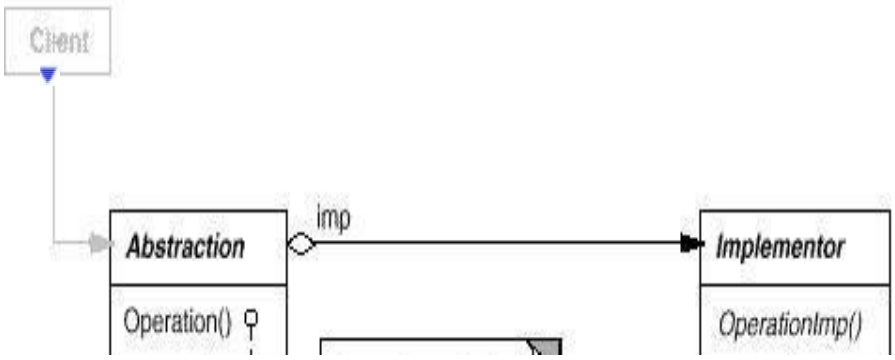
1. To avoid a permanent binding between an abstraction and its implementation. This might be the case, for example, when the implementation must be selected or switched at run-time.

2. both the abstractions and their implementations should be extensible by subclassing. In this case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently.

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

4. (C++) To hide the implementation of an abstraction completely from clients. In C++ the representation of a class is visible in the class interface.

5. To share an implementation among multiple objects (perhaps using reference counting), and this fact should be hidden from the client. A simple example is Coplien's String class [Cop92], in which multiple objects can share the same string representation (StringRep).



Participants

1. **Abstraction (Window)**

   -defines the abstraction's interface.

   -maintains a reference to an object of type Implementor.

2. **RefinedAbstraction (IconWindow)**

   Extends the interface defined by Abstraction.

3. **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.

4. **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:

1.Decoupling interface and implementation.

Decoupling Abstraction and Implementor also eliminates compile-time dependencies on the implementation. Changing an implementation class doesn't require recompiling the Abstraction class and its clients. This property is essential when you must ensure binary compatibility between different versions of a class library.

Furthermore, this decoupling encourages layering that can lead to a better-structured system. The high-level part of a system only has to know about Abstraction and Implementer.

2. Improved extensibility. You can extend the Abstraction and Implementer hierarchies independently.

3.Hiding implementation details from clients. You can shield clients from implementation details, like the sharing of implementer objects and the accompanying reference count mechanism (if any).

Implementation

Consider the following implementation issues when applying the Bridge pattern:

1.Only one Implementor. In situations where there's only one implementation, creating an abstract Implementor class isn't necessary. This is a degenerate case of the Bridge pattern; there's a one-to-one relationship between Abstraction and Implementor. Nevertheless, this separation is still useful when a change in the implementation of a class must not affect its existing clients—that is, they shouldn't have to be recompiled, just relinked.

2.creating the right Implementor object.

1 If Abstraction knows about all ConcreteImplementor classes, then it can instantiate one of them in its constructor; it can decide between them based on parameters passed to its constructor. If, for example, a collection class supports multiple implementations, the decision can be based on the size of the collection.

A linked list implementation can be used for small collections and a hash table for larger ones

Another approach is to choose a default implementation initially and change it later according to usage. For example, if the collection grows bigger than a certain threshold, then it switches its implementation to one that's more appropriate for a large number of items.
It's also possible to delegate the decision to another object altogether.

3.Sharing implementers. Coplien illustrates how the Handle/Body idiom in C++ can be used to share implementations among several objects [Cop92]. The Body stores a reference count that the Handle class increments and decrements. The code for assigning handles with shared bodies has the following general for

```
Handle& Handle::operator= (const Handle& other)  {
other._body->Ref();
_body->Unref();

if (_body->RefCount() == 0) {
delete _body;
}
_body = other._body;

return *this;
}
```

4.Using multiple inheritance. You can use multiple inheritance in C++ to combine an interface with its implementation [Mar91]. For example, a class can inherit publicly from Abstraction and privately from a ConcreteImplementor. But because this approach relies on static inheritance, it binds an implementation

permanently to its interface. Therefore you can't implement a true Bridge with multiple inheritance—at least not in C++.

Sample Code

The following C++ code implements the Window/WindowImp 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 Point& 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

};
```

Window maintains a reference to a WindowImp, the abstract class that declares an interface to the underlying windowing system.

```
class WindowImp {

public:

virtual void ImpTop() = 0;

virtual void ImpBottom() = 0;

virtual void ImpSetExtent(const Point&) = 0;

virtual void ImpSetOrigin(const Point&) = 0;

virtual void DeviceRect(Coord, Coord, Coord, Coord) = 0;
virtual void DeviceText(const char*, Coord, Coord) = 0;
virtual void DeviceBitmap(const char*, Coord, Coord) = 0;

//lots more functions for drawing on windows...

protected:

WindowImp();

};
```

Known uses

libg++ [Lea88] defines classes that implement common data structures, such as Set, LinkedSet, HashSet, LinkedList, and HashTable. Set is an abstract class that defines a set abstraction, while LinkedList and HashTable are concrete implementors for a linked list and a hash table, respectively. LinkedSet and HashSet are Set implementors that bridge between Set and their concrete counterparts LinkedList and HashTable. This is an example of a degenerate bridge, because there's no abstract Implementor class.

NeXT's AppKit [Add94] uses the Bridge pattern in the implementation and display of graphical images. An image can be represented in several different ways. The optimal display of an image depends on the properties of a display device, specifically its color capabilities and its resolution. Without help from AppKit, developers would have to determine which implementation to use under various circumstances in every application.

Related Patterns

An Abstract Factory (99) can create and configure a particular Bridge.

The Adapter (157) pattern is geared toward making unrelated classes work together. It is usually applied to systems after they're designed. Bridge, on the other hand, is used up-front in a design to let abstractions and implementations vary independently.

Composite

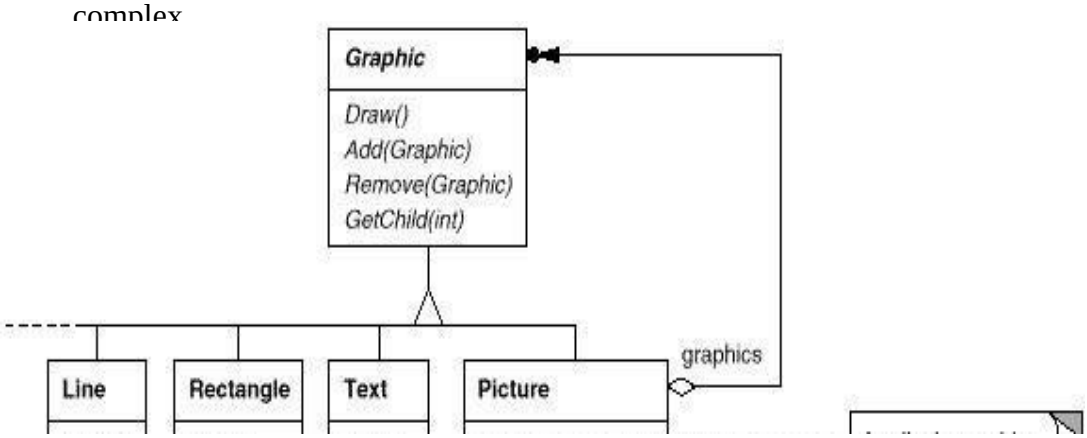
Intent

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. A simple implementation could define classes for graphical primitives such as Text and Lines plus other classes that act as containers for these primitives.

But there's a problem with this approach: Code that uses these classes must treat primitive and container objects differently, even if most of the time the user treats them identically. Having to distinguish these objects makes the application more complex.

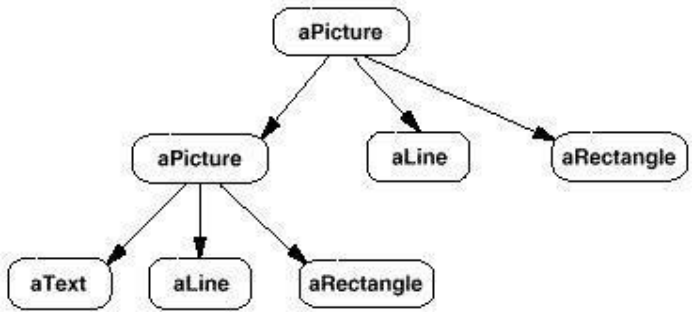


The key to the Composite pattern is an abstract class that represents both primitives and their containers. For the graphics system, this class is Graphic. Graphic declares operations like Draw that are specific to graphical objects.
It also declares operations that all composite objects share, such as operations for accessing and managing its children.

The subclasses Line, Rectangle, and Text (see preceding class diagram) define primitive graphical objects. These classes implement Draw to draw lines, rectangles, and text, respectively. Since primitive graphics have no child graphics, none of these subclasses implements child-related operations.

The Picture class defines an aggregate of Graphic objects. Picture implements Draw to call Draw on its children, and it implements child-related operations accordingly. Because the Picture interface conforms to the Graphic interface, Picture objects can compose other Pictures recursively.

The following diagram shows a typical composite object structure of recursively composed Graphic objects:



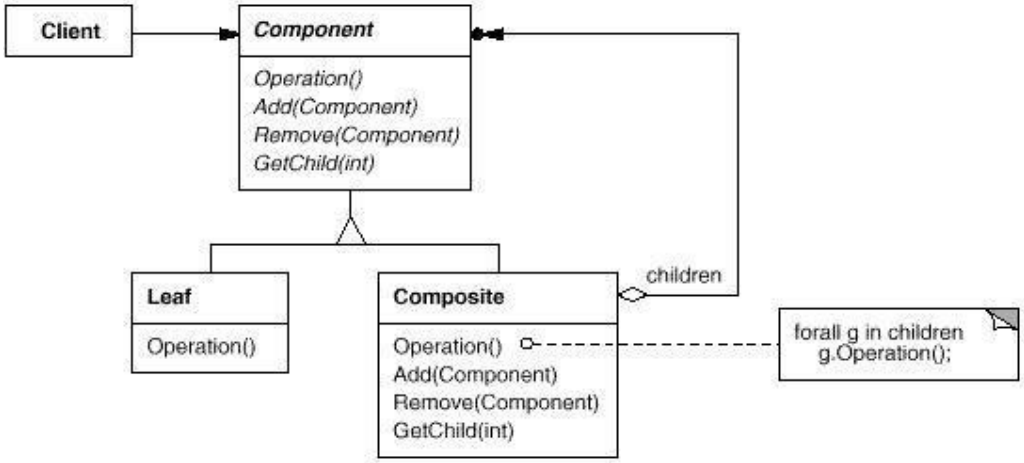
Applicability

Use the Composite pattern when

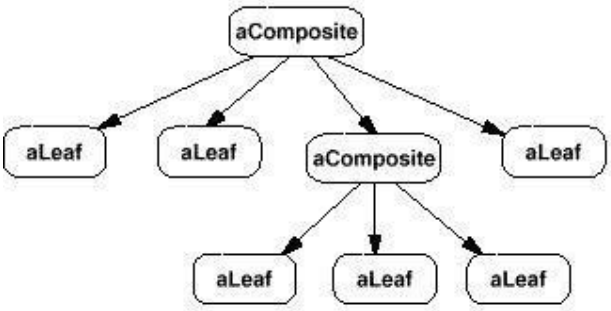
1. To represent part-whole hierarchies of objects.

2. 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



A typical Composite object structure might look like this:



Participants

1. 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.

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

- represents leaf objects in the composition. A leaf has no children.

- defines behavior for primitive objects in the composition.

3. Composite (Picture)

- defines behavior for components having children.

- stores child components.

- implements child-related operations in the Component interface.

4. 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

1. 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 so on recursively. Wherever client code expects a primitive object, it can also take a composite object.

2. 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. This simplifies client code, because it avoids having to write tag-and-case-statement-style functions over the classes that define the composition.

3.makes it easier to add new kinds of components. Newly defined Composite or Leaf subclasses work automatically with existing structures and client code. Clients don't have to be changed for new Component classes.

4. can make your design overly general. The disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite.

Implementation

There are many issues to consider when implementing the Composite pattern:
1.Explicit parent references. Maintaining references from child components to their parent can simplify the traversal and management of a composite structure. The parent reference simplifies moving up the structure and deleting a component. Parent references also help support the Chain of Responsibility (251) pattern.

The usual place to define the parent reference is in the Component class. Leaf and Composite classes can inherit the reference and the operations that manage it.

The easiest way to ensure this is to change a component's parent only when it's being added or removed from a composite. If this can be implemented once in the Add and Remove operations of the Composite class, then it can be inherited by all the subclasses, and the invariant will be maintained automatically.

2.Sharing components. It's often useful to share components, for example, to reduce storage requirements. But when a component can have no more than one parent, sharing components becomes difficult.

A possible solution is for children to store multiple parents. But that can lead to ambiguities as a request propagates up the structure.

4.Maximizing the Component interface. One of the goals of the Composite pattern is to make clients unaware of the specific Leaf or Composite classes they're using. To attain this goal, the Component class should define as many common operations for Composite and Leaf classes as possible. The Component class usually provides default implementations for these operations, and Leaf and Composite subclasses will override them.

if we view a Leaf as a Component that never has children, then we can define a default operation for child access in the Component class that never returns any children. Leaf classes can use the default implementation, but Composite classes will reimplement it to return their children.

4.Declaring the child management operations. Although the Composite class implements the Add and Remove operations for managing children, an important issue in the Composite pattern is which classes declare these operations in the Composite class hierarchy.

The decision involves a trade-off between safety and transparency:

Defining the child management interface at the root of the class hierarchy gives you transparency, because you can treat all components uniformly. It costs you safety, however, because clients may try to do meaningless things like add and remove objects from leaves.

Defining child management in the Composite class gives you safety, because any attempt to add or remove objects from leaves will be caught at compile-time in a statically typed language like C++. But you lose transparency, because leaves and composites have different interfaces.

One approach is to declare an operation Composite* GetComposite() in the Component class. Component provides a default operation that returns a null

pointer. The Composite class redefines this operation to return itself through the this pointer:

```
}

class Composite;

class Component {
public:
virtual Composite* GetComposite() { return 0; }
};

class Composite : public Component {
public:
void Add(Component*);
// ...
virtual Composite* GetComposite() { return this; }
};

class Leaf : public Component {
// ...
};
```

GetComposite lets you query a component to see if it's a composite. You can perform Add and Remove safely on the composite it returns.

```
Composite* aComposite = new Composite;
Leaf* aLeaf = new Leaf;

Component* aComponent;
Composite* test;
```

```
aComponent = aComposite;
if (test = aComponent->GetComposite()) {
test->Add(new Leaf);
}

aComponent = aLeaf;

if (test = aComponent->GetComposite()) {
test->Add(new Leaf); // will not add leaf

}
```

5. Should Component implement a list of Components?

But putting the child pointer in the base class incurs a space penalty for every leaf, even though a leaf never has children. This is worthwhile only if there are relatively few children in the structure.

6. Child ordering. Many designs specify an ordering on the children of Composite. In the earlier Graphics example, ordering may reflect front-to-back ordering. If Composites represent parse trees, then compound statements can be instances of a Composite whose children must be ordered to reflect the program.

When child ordering is an issue, you must design child access and management interfaces carefully to manage the sequence of children. The Iterator (289) pattern can guide you in this.

7. Caching to improve performance. If you need to traverse or search compositions frequently, the Composite class can cache traversal or search information about its children. The Composite can cache actual results or just information that lets it short-circuit the traversal or search.

Changes to a component will require invalidating the caches of its parents. This works best when components know their parents. So if you're using caching, you need to define an interface for telling composites that their caches are invalid.

8. Who should delete components? In languages without garbage collection, it's usually best to make a Composite responsible for deleting its children when it's destroyed. An exception to this rule is when Leaf objects are immutable and thus can be shared.

9. What's the best data structure for storing components? Composites may use a variety of data structures to store their children, including linked lists, trees, arrays, and hash tables. The choice of data structure depends (as always) on efficiency. In fact, it isn't even necessary to use a general-purpose data structure at all.

Sample Code

Equipment such as computers and stereo components are often organized into part-whole or containment hierarchies. For example, a chassis can contain drives and planar boards, a bus can contain cards, and a cabinet can contain chassis, buses, and so forth. Such structures can be modeled naturally with the Composite pattern.

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* CreateIterator();
    protected:
    Equipment(const char*);
    private:
    const char* _name;
};
```

Equipment declares operations that return the attributes of a piece of equipment, like its power consumption and cost. Subclasses implement these operations for specific kinds of equipment. Equipment also declares a CreateIterator operation that returns an Iterator (see Appendix C) for accessing its parts. The default implementation for this operation returns a NullIterator, which iterates over the empty set.

Subclasses of Equipment might include Leaf classes that represent disk drives, integrated circuits, and switches:

```
class FloppyDisk : public Equipment {
public:
FloppyDisk(const char*);
virtual ~FloppyDisk();

virtual Watt Power();
virtual Currency NetPrice();
virtual Currency DiscountPrice();
```

};

CompositeEquipment is the base class for equipment that contains other equipment.

It's also a subclass of Equipment.

```
class CompositeEquipment : public Equipment {

public:

virtual ~CompositeEquipment();


virtual Watt Power();

virtual Currency NetPrice();

virtual Currency DiscountPrice();


virtual void Add(Equipment*);

virtual void Remove(Equipment*);

virtual Iterator* CreateIterator();


protected:

CompositeEquipment(const char*);

private:

List _equipment;

};
```

Known Uses

Examples of the Composite pattern can be found in almost all object-oriented systems. The original View class of Smalltalk Model/View/Controller [KP88] was a Composite, and nearly every user interface toolkit or framework has followed in its steps, including

ET++ (with its VObjects [WGM88]) and InterViews (Styles [LCI+92], Graphics [VL88], and Glyphs [CL90]). Release 4.0 of Smalltalk-80 revised Model/View/Controller with a VisualComponent class that has subclasses View and CompositeView.

The RTL Smalltalk compiler framework [JML92] uses the Composite pattern extensively. RTLExpression is a Component class for parse trees. It has subclasses, such as BinaryExpression, that contain child RTLExpression objects. These classes define a composite structure for parse trees. RegisterTransfer is the Component class for a program's intermediate Single Static Assignment (SSA) form. Leaf subclasses of RegisterTransfer define different static assignments such as

1. primitive assignments that perform an operation on two registers and assign the result to a third;

2. an assignment with a source register but no destination register, which indicates that the register is used after a routine returns; and

3. an assignment with a destination register but no source, which indicates that the register is assigned before the routine starts.

Another subclass, RegisterTransferSet, is a Composite class for representing assignments that change several registers at once.

Related Patterns

Often the component-parent link is used for a Chain of Responsibility (251).

Decorator (196) is often used with Composite. When decorators and composites are used together, they will usually have a common parent class. So decorators will have to support the Component interface with operations like Add, Remove, and GetChild.

Flyweight (218) lets you share components, but they can no longer refer to their parents.

Iterator (289) can be used to traverse composites.

Visitor (366) localizes operations and behavior that would otherwise

Decorator

Intent

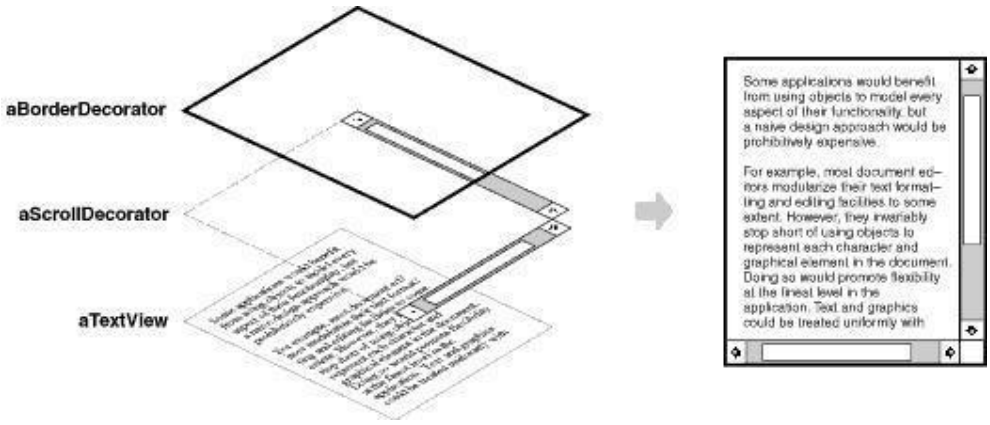
Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Also Known As

Wrapper

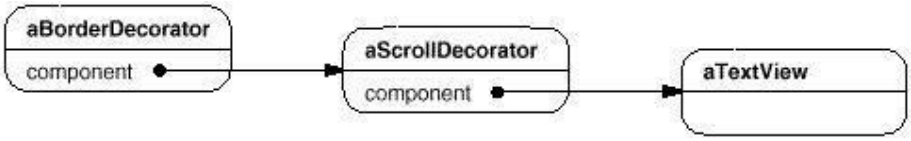
Motivation

A more flexible approach is to enclose the component in another object that adds the border. The enclosing object is called a decorator. The decorator conforms to the interface of the component it decorates so that its presence is transparent to the component's clients. The decorator forwards requests to the component and may perform additional actions (such as drawing a border) before or after forwarding. Transparency lets you nest decorators recursively, thereby allowing an unlimited number of added responsibilities.

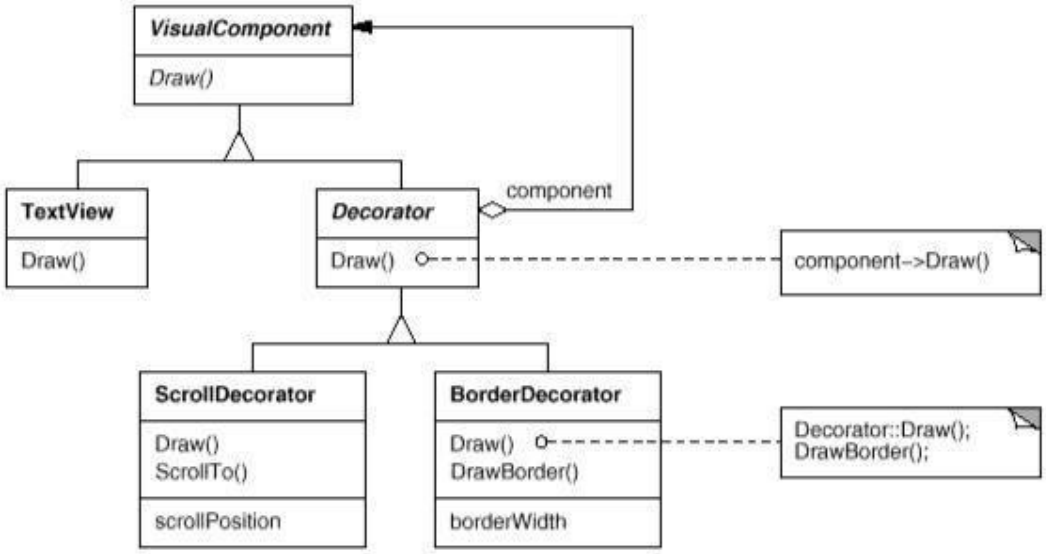


For example, suppose we have a TextView object that displays text in a window. TextView has no scroll bars by default, because we might not always need them. When we do, we can use a ScrollDecorator to add them. Suppose we also want to add a thick black border around the TextView. We can use a BorderDecorator to add this as well. We simply compose the decorators with the TextView to produce the desired result.

The following object diagram shows how to compose a TextView object with BorderDecorator and ScrollDecorator objects to produce a bordered, scrollable text view:



The ScrollDecorator and BorderDecorator classes are subclasses of Decorator, an abstract class for visual components that decorate other visual components.



VisualComponent is the abstract class for visual objects. It defines their drawing and event handling interface. Note how the Decorator class simply forwards draw requests to its component, and how Decorator subclasses can extend this operation.

Decorator subclasses are free to add operations for specific functionality. For example, ScrollDecorator's ScrollTo operation lets other objects scroll the interface if they know there happens to be a ScrollDecorator object in the interface.

APPLICABILITY

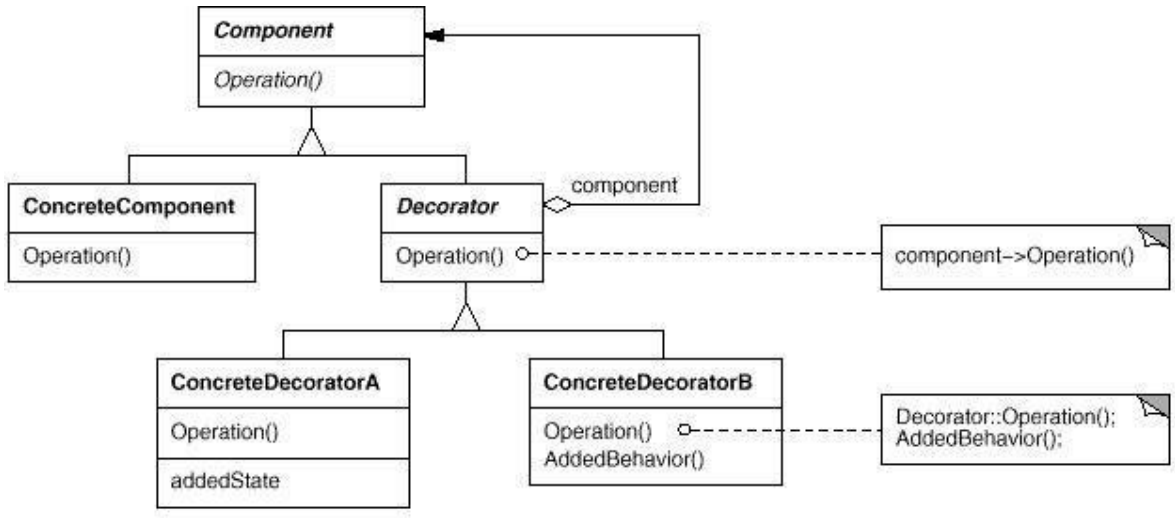
Use Decorator

1.To add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.

2. for responsibilities that can be withdrawn.

3. when extension by subclassing 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 subclassing.

Structure



Participants

Component (VisualComponent)

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

ConcreteComponent (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.

ConcreteDecorator (BorderDecorator, ScrollDecorator)

adds responsibilities to the component.

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:

1.More flexibility than static inheritance. The Decorator pattern provides a more flexible way to add responsibilities to objects than can be had with static (multiple) inheritance. With decorators, responsibilities can be added and removed at run-time simply by attaching and detaching them. In contrast, inheritance requires creating a new class for each additional responsibility (e.g., BorderedScrollableTextView, BorderedTextView).

This gives rise to many classes and increases the complexity of a system. Furthermore, providing different Decorator classes for a specific Component class lets you mix and match responsibilities.

Decorators also make it easy to add a property twice. For example, to give a TextView a double border, simply attach two BorderDecorators. Inheriting from a Border class twice is error-prone at best.

2. **Avoids feature-laden classes high up in the hierarchy**. Decorator offers a pay-as-you-go approach to adding responsibilities. Instead of trying to support all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects. Functionality can be composed from simple pieces. As a result, an application needn't pay for features it doesn't use

3. **A decorator and its component aren't identical**. A decorator acts as a transparent enclosure. But from an object identity point of view, a decorated component is not identical to the component itself. Hence you shouldn't rely on object identity when you use decorators.

4. **Lots of little objects**. A design that uses Decorator often results in systems composed of lots of little objects that all look alike. The objects differ only in the way they are interconnected, not in their class or in the value of their variables. Although these systems are easy to customize by those who understand them, they can be hard to learn and debug.

Implementation

Several issues should be considered when applying the Decorator pattern:

1. **Interface conformance.** A decorator object's interface must conform to the interface of the component it decorates. ConcreteDecorator classes must therefore inherit from a common class (at least in C++).

2. **Omitting the abstract Decorator class**. There's no need to define an abstract Decorator class when you only need to add one responsibility. That's often the case when you're dealing with an existing class hierarchy rather than

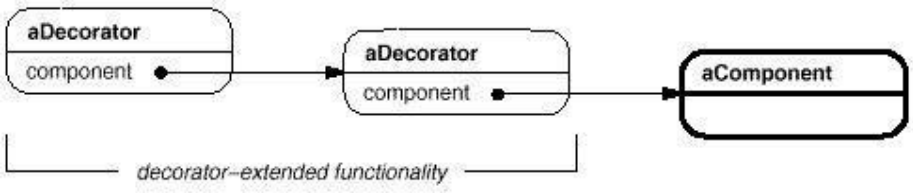
designing a new one. In that case, you can merge Decorator's responsibility for forwarding requests to the component into the ConcreteDecorator.

Keeping Component classes lightweight. To ensure a conforming interface, components and decorators must descend from a common Component class. It's important to keep this common class lightweight; that is, it should focus on defining an interface, not on storing data.

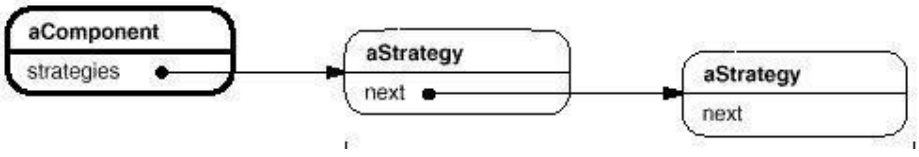
3. Changing the skin of an object versus changing its guts. We can think of a decorator as a skin over an object that changes its behavior. An alternative is to change the object's guts. The Strategy (349) pattern is a good example of a pattern for changing the guts.

Strategies are a better choice in situations where the Component class is intrinsically heavyweight, thereby making the Decorator pattern too costly to apply. In the Strategy pattern, the component forwards some of its behavior to a separate strategy object. The Strategy pattern lets us alter or extend the component's functionality by replacing the strategy object.

Since the Decorator pattern only changes a component from the outside, the component doesn't have to know anything about its decorators; that is, the decorators are transparent to the component:



With strategies, the component itself knows about possible extensions. So it has to reference and maintain the corresponding strategies:



The Strategy-based approach might require modifying the component to accommodate new extensions. On the other hand, a strategy can have its own specialized interface, whereas a decorator's interface must conform to the component's. A strategy for rendering a border, for example, need only define the interface for rendering a border (DrawBorder, GetWidth, etc.), which means that the strategy can be lightweight even if the Component class is heavyweight.

Sample Code

The following code shows how to implement user interface decorators in C++. We'll assume there's a Component class called VisualComponent.

```
class VisualComponent {
public:
    VisualComponent();
```



```
    virtual void Draw();
    virtual void Resize();
    // ...
};
```

We define a subclass of VisualComponent called Decorator, which we'll subclass to obtain different decorations.

```
class Decorator : public VisualComponent {
```

```
public:
    Decorator(VisualComponent*);

    virtual void Draw();
    virtual void Resize();
        ...
private:
    VisualComponent* _component;
};
```

Decorator decorates the VisualComponent referenced by the _component instance variable, which is initialized in the constructor. For each operation in VisualComponent's interface, Decorator defines a default implementation that passes the request on to _component:

```
void Decorator::Draw () {
    _component->Draw();
}

void Decorator::Resize () {
    _component->Resize();
}
```

Subclasses of Decorator define specific decorations. For example, the class BorderDecorator adds a border to its enclosing component. BorderDecorator is a subclass of Decorator that overrides the Draw operation to draw the border.

BorderDecorator also defines a private DrawBorder helper operation that does the drawing. The subclass inherits all other operation implementations from Decorator.

```
class BorderDecorator : public Decorator {
public:
    BorderDecorator(VisualComponent*, int borderWidth);


    virtual void Draw();
private:
    void DrawBorder(int);
private:
    int _width;
};

void BorderDecorator::Draw () {
```

```
        Decorator::Draw();
        DrawBorder(_width);
    }

        new ScrollDecorator(textView), 1
    )
);
```

Because Window accesses its contents through the VisualComponent interface, it's unaware of the decorator's presence. You, as the client, can still keep track of the text view if you have to interact with it directly, for example, when you need to invoke operations that aren't part of the VisualComponent interface. Clients that rely on the component's identity should refer to it directly as well.

Known Uses

Streams are a fundamental abstraction in most I/O facilities. A stream can provide an interface for converting objects into a sequence of bytes or characters. That lets us transcribe an object to a file or to a string in memory for retrieval later. A straightforward way to do this is to define an abstract Stream class with subclasses MemoryStream and FileStream. But suppose we also want to be able to do the following:

1. Compress the stream data using different compression algorithms (run-length encoding, Lempel-Ziv, etc.).

2. Reduce the stream data to 7-bit ASCII characters so that it can be transmitted over an ASCII communication channel.

The Decorator pattern gives us an elegant way to add these responsibilities to streams. The diagram below shows one solution to the problem:

The Stream abstract class maintains an internal buffer and provides operations for storing data onto the stream (PutInt, PutString). Whenever the buffer is full, Stream calls the abstract operation HandleBufferFull, which does the actual data transfer. The FileStream version of this operation overrides this operation to transfer the buffer to a file.

The key class here is StreamDecorator, which maintains a reference to a component stream and forwards requests to it. StreamDecorator subclasses override HandleBufferFull and perform additional actions before calling StreamDecorator's HandleBufferFull operation.

For example, the CompressingStream subclass compresses the data, and the ASCII7Stream converts the data into 7-bit ASCII. Now, to create a FileStream that compresses its data and converts the compressed binary data to 7-bit ASCII, we decorate a FileStream with a CompressingStream and an ASCII7Stream:

```
Stream* aStream = new CompressingStream(

new ASCII7Stream(

new FileStream("aFileName")
)
);
aStream->PutInt(12);
```

```
aStream->PutString("aString");
```

Related Patterns

Adapter (157): A decorator is different from an adapter in that a decorator only changes an object's responsibilities, not its interface; an adapter will Composite (183): A decorator can be viewed as a degenerate composite with only one component. However, a decorator adds additional responsibilities—it isn't intended for object aggregation.

Strategy (349): 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.

Façade

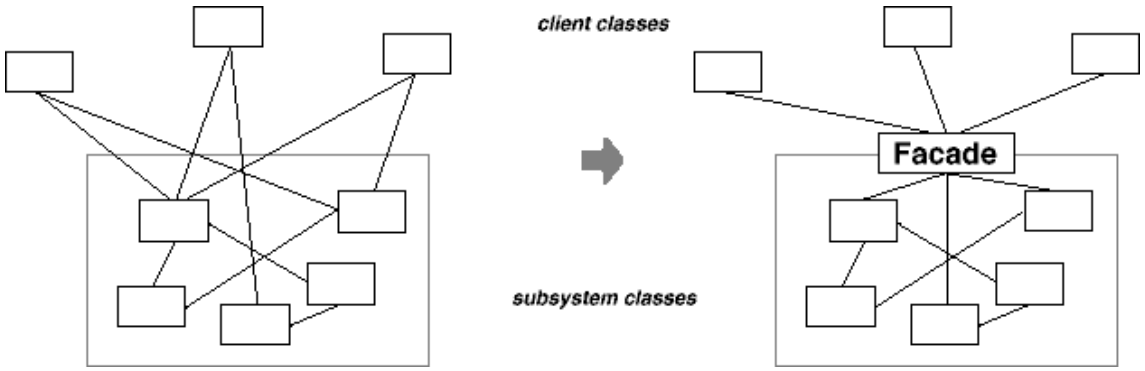
Intent

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.
-

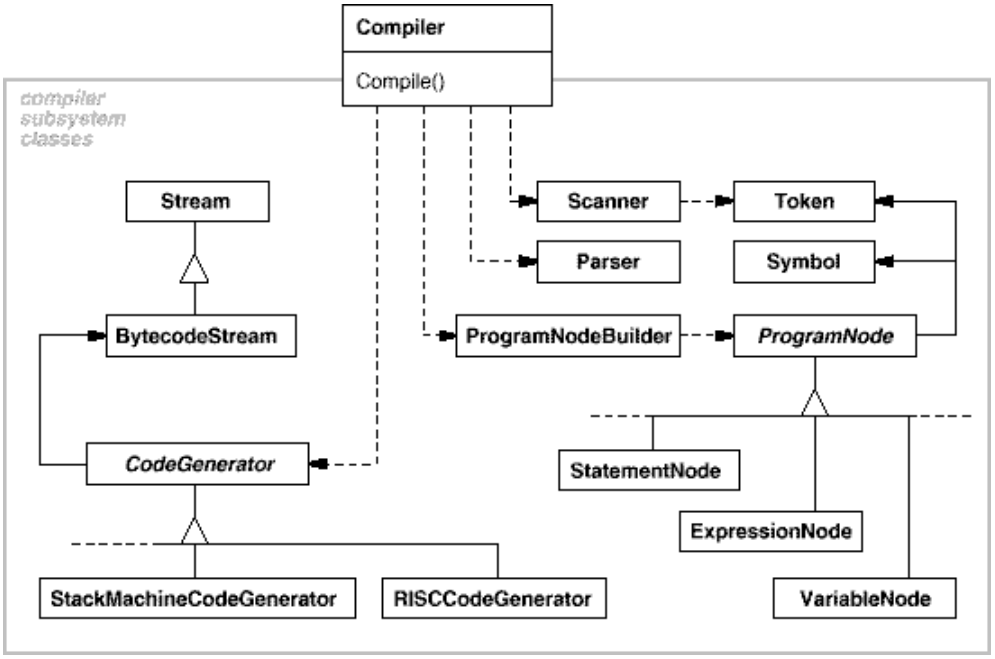
- One way to achieve this goal is to introduce a **facade** object that provides a single, simplified interface to the more general facilities of a subsystem.



- Consider for example a programming environment that gives applications access toitscompilersubsystem.Thissubsystemcontainsclassessuchas Scanner, Parser, Program Node, BytecodeStream, and ProgramNodeBuilder that implementthecompiler.

- To provide a higher-level interface that can shield clients from these classes, the compiler subsystem also includes a Compiler class. This class defines a unified interface to the compiler's functionality.

- The Compiler class acts as a facade: It offers clients a single, simple interface to the compiler subsystem. It glues together the classes that implement compiler functionality without hiding them completely. The compiler facade makes life easier for most programmers without hiding the lower-level functionality from the few that need it.



Applicability

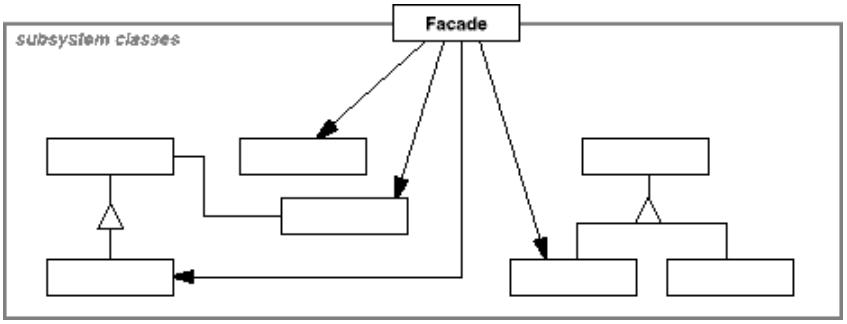
Use the Facade pattern when

- you want to provide a simple interface to a complex subsystem. Subsystems often get more complex as they evolve. Most patterns, when applied, result in more and smaller classes. This makes the subsystem more reusable and easier to customize, but it also becomes harder to use for clients that don't need to customize it. A facade can provide a simple default view of the subsystem that is good enough for most clients. Only clients needing more customizability will need to look beyond the facade.

- There are manydependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystemfrom clients and other subsystems, thereby promoting subsystem independence and portability.

- You want to layer your subsystems. Use a facade to define an entry point to each subsystem level. If subsystems are dependent, then you can simplify the dependencies between them by making them communicate with each other solely through their facades.

Structure



Participants

- **Facade** (Compiler)
  - knows which subsystem classes are responsible for a request.
  - delegates 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

- Clientscommunicatewiththesubsystembysendingrequeststo 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 subsystemobjects directly.

Consequences

The Facade pattern offers the following benefits:

- It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.
- It promotes weak coupling between the subsystem and its clients. Often the components in a subsystemare strongly coupled. Weak coupling lets you vary the componentsofthesubsystemwithoutaffectingitsclients.
- Facades help layer a system and the dependencies between objects. They can eliminate complexorcirculardependencies.Thiscanbeanimportantconsequence when the client and the subsystem are implemented independently.

- Reducing compilation dependencies is vital in large software systems. You want to save time by minimizing recompilation when subsystem classes change. Reducing compilation dependencies with facades can limit the recompilation needed for a small change in an important subsystem. A facade can also simplify porting systems to other platforms, because it's less likely that building one subsystem requires building all others.

- It doesn't prevent applications from using subsystem classes if they need to. Thus you can choose between ease of use and generality.

Implementation

Consider the following issues when implementing a facade:

- ***Reducing client-subsystem coupling***. The coupling between clients and the subsystem can be reduced even further by making Facade an abstract class with concrete subclasses fordifferent implementationsofa subsystem.Then clients can communicate with the subsystem through the interface of the abstract Facade class. This abstract coupling keeps clients from knowing which implementation of a subsystem is used.

  An alternative to sub classing is to configure a Facade object with different subsystem objects. To customize the facade, simply replace one or more of its subsystem objects.

- ***Public versus private subsystem classes***. A subsystem is analogous to a class in that both have interfaces, and both encapsulate something—a class encapsulatesstateandoperations,whileasubsystemencapsulatesclasses.
  1. The public interface to a subsystem consists of classes that all clients can access; the private interface is just for subsystem extenders.
  2. The Facade class is part of the public interface, of course, but it's not the only part. Other subsystem classes are usually public as well. For

example, the classes Parser and Scanner in the compiler subsystem are part of the public interface.

3. Making subsystem classes private would be useful, but few object-oriented languagessupportit.BothC++andSmalltalktraditionallyhave had a global name space for classes. Recently, however, the C++ standardization committee added name spaces to the language [Str94], which will let you expose just the public subsystem classes.

Sample Code

Let's take a closer look at how to put a facade on a compiler subsystem.

The compiler subsystem defines a {BytecodeStream} class that implements a stream of Bytecodeobjects. ABytecodeobject encapsulatesabytecode, whichcanspecify machine instructions. The subsystem also defines a Token class for objects that encapsulate tokens in the programming language.

The Scanner class takes a stream of characters and produces a stream of tokens, one token at a time.

```
class Scanner { public:
    Scanner(istream&); virtual ~Scanner();

    virtual Token& Scan(); private:
    istream& _inputStream;
};
```

The class Parser uses a ProgramNodeBuilder to construct a parse tree from a Scanner's tokens.

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

    virtual void Parse(Scanner&, ProgramNodeBuilder&);
};
```

Parser calls back on ProgramNodeBuilder to build the parse tree incrementally. These classes interact according to the Builder (110) pattern.

```
class ProgramNodeBuilder {

public:
    ProgramNodeBuilder();

    virtual ProgramNode* NewVariable( const char* variableName
    ) const;

    virtual ProgramNode* NewAssignment(
        ProgramNode* variable, ProgramNode* expression
    ) const;
```

```
    virtual ProgramNode* NewReturnStatement( ProgramNode* value
    ) const;


    virtual ProgramNode* NewCondition( ProgramNode* condition,
        ProgramNode* truePart, ProgramNode* falsePart
    ) const;
    // ...


    ProgramNode* GetRootNode(); private:
    ProgramNode* _node;
};
```

The parse tree is made up of instances of ProgramNode subclasses such as StatementNode, ExpressionNode, and so forth. The ProgramNode hierarchy is an example of the Composite (183) pattern. ProgramNode defines an interface for manipulating the program node and its children, if any.

```
class ProgramNode { public:
    // program node manipulation
    virtual void GetSourcePosition(int& line, int& index);
```

```
    // ...


    // child manipulation

    virtual void Add(ProgramNode*); virtual void Remove(ProgramNode*);
    // ...


    virtual void Traverse(CodeGenerator&);


protected:
    ProgramNode();
};
```

The Traverse operation takes a CodeGenerator object. ProgramNode subclasses use this object to generate machine code in the form of Bytecode objects on a BytecodeStream. The class CodeGenerator is a visitor (see Visitor (366)).

```
class CodeGenerator { public:
    virtual void Visit(StatementNode*); virtual void Visit(ExpressionNode*);
    // ... protected:
    CodeGenerator(BytecodeStream&); protected:
    BytecodeStream& _output;
```

```
};
```

CodeGenerator has subclasses, for example, StackMachineCodeGenerator and RISCCodeGenerator, that generate machine code for different hardware architectures.

Each subclass of ProgramNode implements Traverse to call Traverse on its child ProgramNode objects. In turn, each child does the same for its children, and so on recursively. For example, ExpressionNode defines Traverse as follows:

```
void ExpressionNode::Traverse (CodeGenerator& cg) { cg.Visit(this);

    ListIterator i(_children);

    for (i.First(); !i.IsDone(); i.Next()) { i.CurrentItem()->Traverse(cg);
    }
}
```

The classes we've discussed so far make up the compiler subsystem. Now we'll introduce a Compiler class, a facade that puts allthese pieces together. Compiler provides a simple interface for compiling source and generating code for a particular machine.

```
class Compiler { public:
```

```
    Compiler();

    virtual void Compile(istream&, BytecodeStream&);
};

void Compiler::Compile (
    istream& input, BytecodeStream& output
) {
    Scanner scanner(input); ProgramNodeBuilder builder; Parser parser;

    parser.Parse(scanner, builder);

    RISCCodeGenerator generator(output); ProgramNode* parseTree =
    builder.GetRootNode(); parseTree->Traverse(generator);
}
```

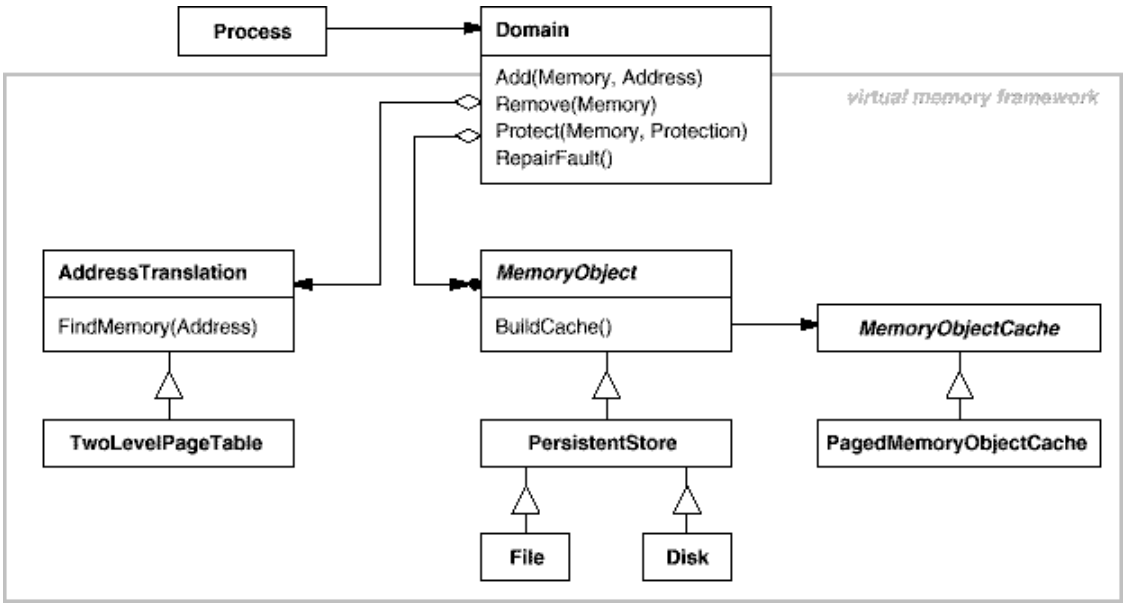
Known Uses

The compiler example in the Sample Code section was inspired by the ObjectWorks\Smalltalk compiler system [Par90].

- In the ET++ application framework [WGM88], an application can have built-in browsing tools for inspecting its objects at run-time. These browsing tools are

implemented in a separate subsystem that includes a Facade class called "ProgrammingEnvironment." This facade defines operations such as InspectObject and InspectClass for accessing the browsers.

- An ET++ application can also forgo built-in browsing support. In that case, Programming Environment implements these requests as null operations; that is, they do nothing. Only the ETProgramming Environment subclass implements these requestswithoperationsthatdisplaythecorresponding browsers..



For example, the virtual memory framework has Domain as its facade. A Domain represents an address space. It provides a mapping between virtual addresses and offsets into memoryobjects, files, or backing store. The main operations on Domain support adding a memoryobject at a particular address, removing a memoryobject, and handling a page fault.

As the preceding diagram shows, the virtual memory subsystem uses the following components internally:

- Memory Object represents a data store.

- Memory Object Cache caches the data of Memory Objects in physical memory. Memory Object Cache is actually a Strategy that localizes the caching policy.

- Address Translation encapsulates the address translation hardware.

The Repair Fault operation is called whenever a page fault interrupt occurs. The Domain finds the memory object at the address causing the fault and delegates the Repair Fault operation to the cache associated with that memory object. Domains can be customized by changing their components.

Related Patterns

Abstract Factory 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 is similar to Facade in that it abstracts functionality of existing classes.

However, Mediator's purpose is to abstract arbitrary communication between colleague objects, often centralizing functionality that doesn't belong in any one of them.

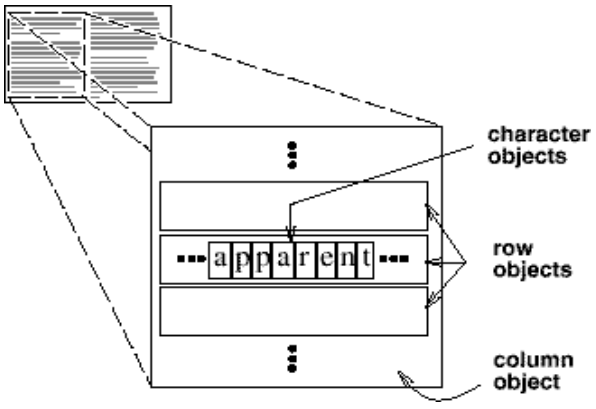
Flyweight

Intent

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

Motivation

Forexample,mostdocument editor implementations have textformattingandediting facilitiesthat are modularized to some extent. Object-oriented document editors typically use objects to represent embedded elements like tables and figures. However, they usually stop short of using an object for each character inthe document, even though doing so would promote flexibility at the finest levels in the application. Characters and embedded elements could then be treated uniformly with respect to how they are drawn and formatted. The application could be extended to support new character sets without disturbing other functionality. The application's object structure could mimic the document's physical structure. The following diagram shows how a document editor can use objects to represent characters.



Thedrawbackofsuch adesignis itscost.

Even moderate-sized documents may require hundreds of thousands of character objects, which will consume lots of memory and may incur unacceptable run-time

overhead. The Flyweight patterndescribes how to share objects to allow their use at fine granularities without prohibitive cost.

A**flyweight** is a shared object that canbe used in multiplecontextssimultaneously. The flyweight acts asanindependentobject ineachcontext—it'sindistinguishable from an instance of the object that's not shared. Flyweights cannot make assumptions about the context in which they operate.

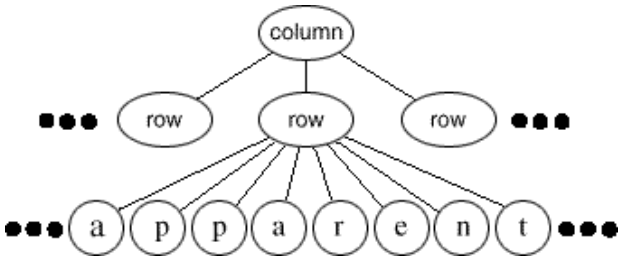
The key concept here is the distinction between **intrinsic** and **extrinsic** state.

- 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.

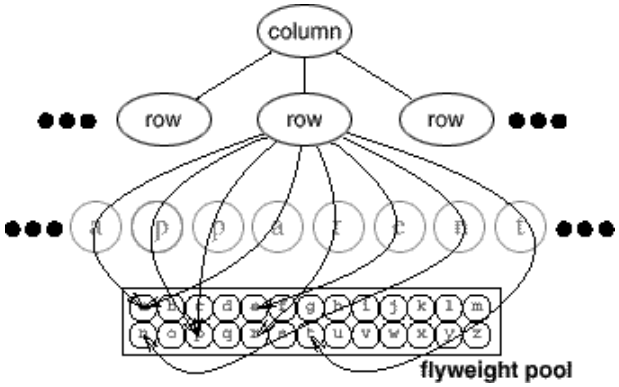
Flyweights model conceptsor entitiesthat are normally too plentiful to represent with objects. For example, a document editor can create a flyweight for each letter of the alphabet.

Each flyweight stores a character code, but its coordinate position in the document and its typographic style can be determined from the text layout algorithms and formatting commands in effect wherever the character appears. The character code is intrinsic state, while the other information is extrinsic.

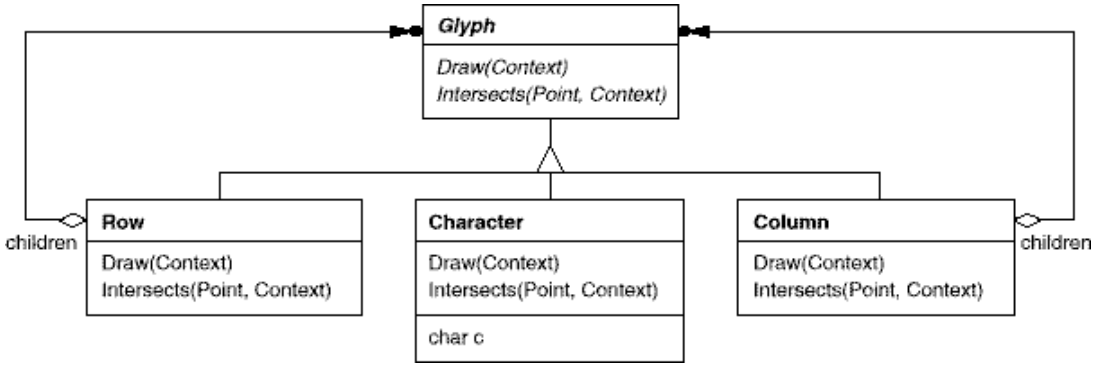
Logically there is an object for every occurrence of a given character in the document:



Physically, however, there is one shared flyweight object per character, and it appears in different contexts in the document structure. Each occurrence of a particular character object refers to the same instance in the shared poolof flyweight objects:



The class structure for these objects is shown next. Glyph is the abstract class for graphical objects, some of which may be flyweights. Operations that may depend on extrinsic state have it passed to them as a parameter. For example, Draw and Intersects must know which context the glyph is in before they can do their job.



A flyweight representing the letter "a" only stores the corresponding character code; it doesn't need to store its location or font. Clients supply the context-dependent informationthattheflyweightneedstodrawitself.Forexample, a Row glyph knows where its children should draw themselves so that they are tiled horizontally. Thus it can pass each child its location in the draw request.

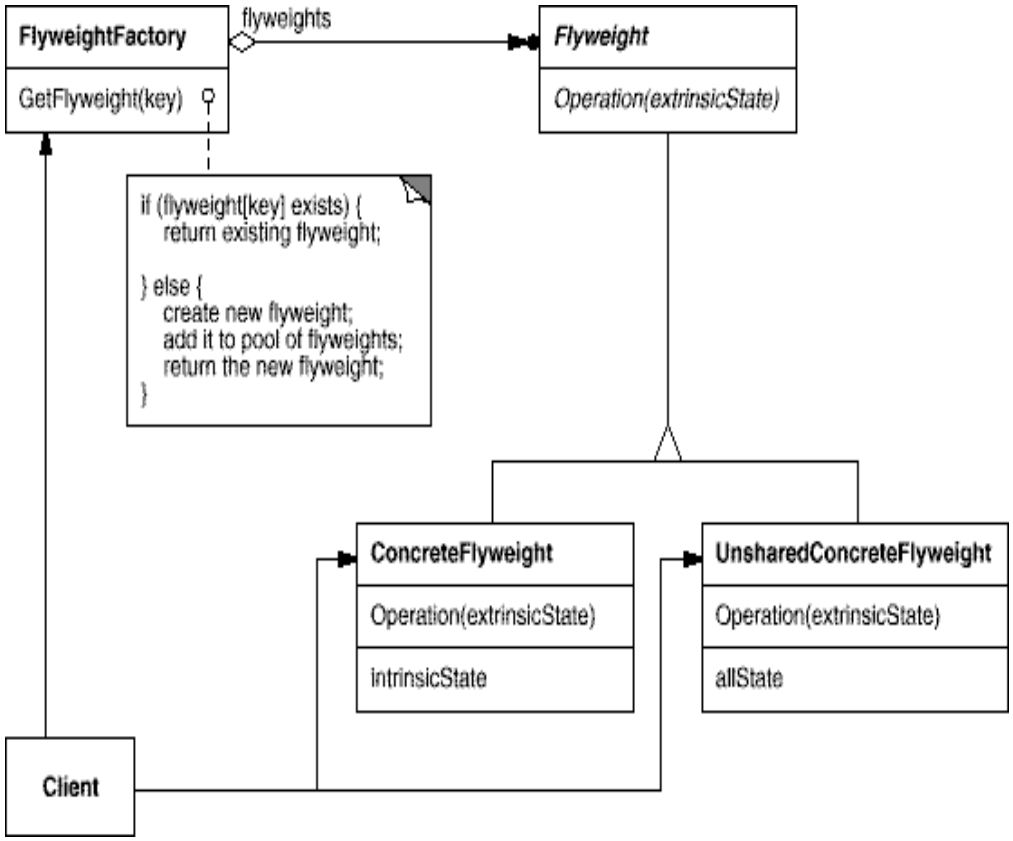
Because the number of different character objects is far less than the number of characters in the document, the total number of objects is substantially less than what a naive implementation would use.

Applicability

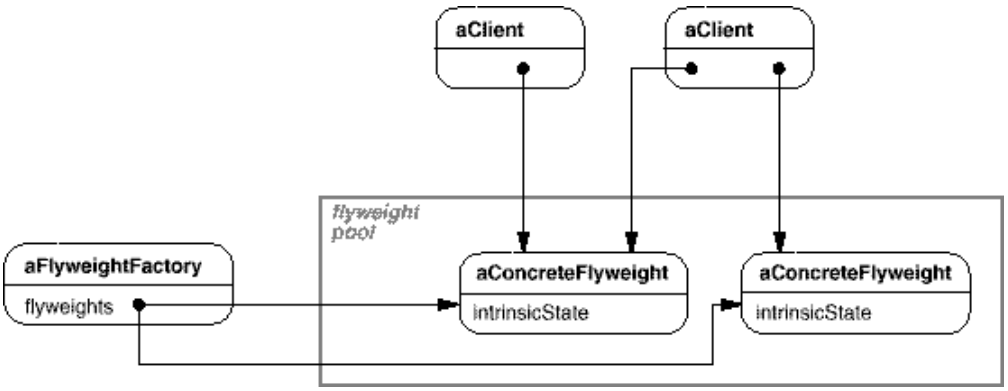
The Flyweight pattern's effectiveness depends heavilyon how and where it's used. Apply the Flyweight pattern when *all* of the following are true:

- 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.
- Theapplicationdoesn't dependonobjectidentity.Sinceflyweightobjects may be shared, identity tests will return true for conceptually distinct objects

Structure



The following object diagram shows how flyweights are shared:



Participants

- **Flyweight**
  - 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. AConcreteFlyweight object mustbesharable. Anystate it stores must be intrinsic; that is, it must be independent of the ConcreteFlyweight object's context.
- **UnsharedConcreteFlyweight** (Row, Column)
  - not all Flyweight subclasses need to be shared. The Flyweight interface *enables* sharing; it doesn't enforce it. It's common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as childrenat some levelin the flyweight object structure (as the Row and Column classes have).
- **FlyweightFactory**
  - creates and manages flyweight objects.
  - ensuresthat flyweightsaresharedproperly. Whenaclient 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 orextrinsic. IntrinsicstateisstoredintheConcreteFlyweight

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

Flyweights may introduce run-time costs associated with transferring, finding, and/or computing extrinsic state, especially if it was formerly stored as intrinsic state. However, such costs are offset by space savings, which increase as more flyweights are shared.

Storage savings are a function of several factors:

- the reduction in the total number of instances that comes from sharing
- the amount of intrinsic state per object
- whether extrinsic state is computed or stored.
- The more flyweights are shared, the greater the storage savings. The savings increase with the amount of shared state.
- The greatest savings occur whenthe objects use substantial quantities of both intrinsic and extrinsic state,and the extrinsic state can be computed rather than stored.
- Then you save on storage intwo ways: Sharing reduces the cost ofintrinsic state, and you trade extrinsic state for computation time.

The Flyweight pattern is often combined with the Composite (183) pattern to represent a hierarchical structure as a graph with shared leaf nodes. A consequence of sharing is that flyweight leaf nodes cannot store a pointer to their parent. Rather, the parent pointer is passed to the flyweight as part of its extrinsic state. This has a major impact on how the objects in the hierarchy communicate with each other.

Implementation

Consider the following issues when implementing the Flyweight pattern:

- *Removing extrinsic state.* The pattern's applicability is determined largely by how easy it is to identify extrinsic state and remove it from shared objects. Removing extrinsic state won't help reduce storage costs iftherearesmany differentkindsofextrinsicstateastherareobjects before sharing. Ideally,

extrinsic state can be computed from a separate object structure, one with far smaller storage requirements.

In our document editor, for example, we can store a map of typographic information in a separate structure rather than store the font and type style with each character object. The map keeps track of runs of characters with the same typographic attributes. When a character draws itself, it receives its typographic attributes as a side-effect of the draw traversal. Because documents normally use just a few different fonts and styles, storing this information externally to each character object is far more efficient than storing it internally.

- *Managing shared objects.* Because objects are shared, clients shouldn't instantiate them directly. Flyweight Factory lets clients locate a particular flyweight. Flyweight Factory objects often use an associative store to let clients look up flyweights of interest. For example, the flyweight factory in the document editor example can keep a table of flyweights indexed by character codes. The manager returns the proper flyweight given its code, creating the flyweight if it does not already exist.

Sharability also implies some form of reference counting or garbage collection to reclaim a flyweight's storage when it's no longer needed. However, neither is necessary if the number of flyweights is fixed and small (e.g., flyweights for the ASCII character set). In that case, the flyweights are worth keeping around permanently.

Sample Code

Returning to our document formatter example, we can define a Glyph base class for flyweight graphical objects. Logically, glyphs are Composites (see Composite (183)) that have graphical attributes and can 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();
};
```

The Character subclass just stores a character code:

```
class Character : public Glyph { public:
    Character(char);

    virtual void Draw(Window*, GlyphContext&); private:
    char _charcode
};
```

To keep from allocating space for a font attribute in every glyph, we'll store the attribute extrinsically in a GlyphContext object.

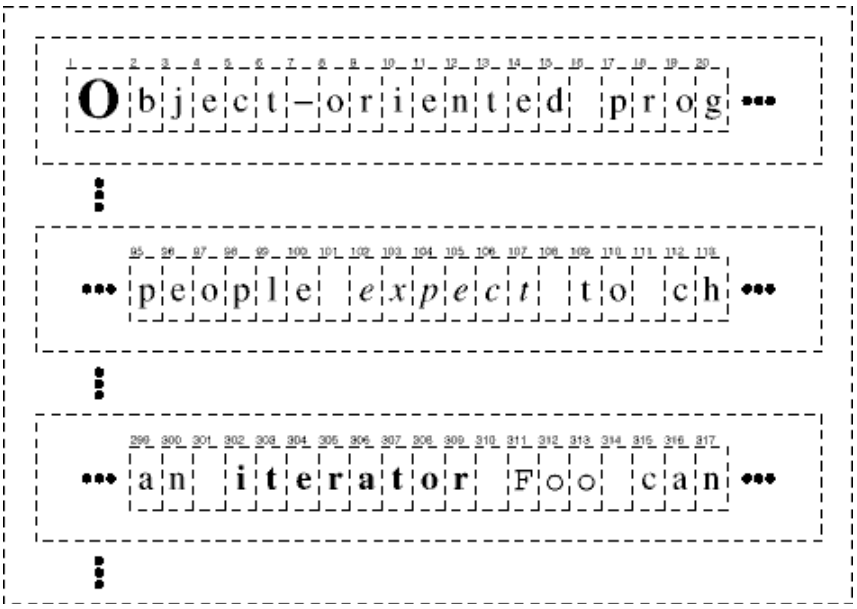
- GlyphContext acts as a repository of extrinsic state. It maintains a compact mapping between a glyph and its font (and any other graphical attributes it might have) in different contexts.
- Any operation that needs to know the glyph's font in a given context will have a GlyphContext instance passed to it as a parameter. The operation can then query the GlyphContext for the font in that context.
- The context depends on the glyph's location in the glyph structure. Therefore Glyph's child iteration and manipulation operations must update the GlyphContext whenever they're used.

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

    virtual void Next(int step = 1); virtual void Insert(int quantity = 1);

    virtual Font* GetFont();
    virtual void SetFont(Font*, int span = 1); private:

    int _index;
    BTree* _fonts;
};
```

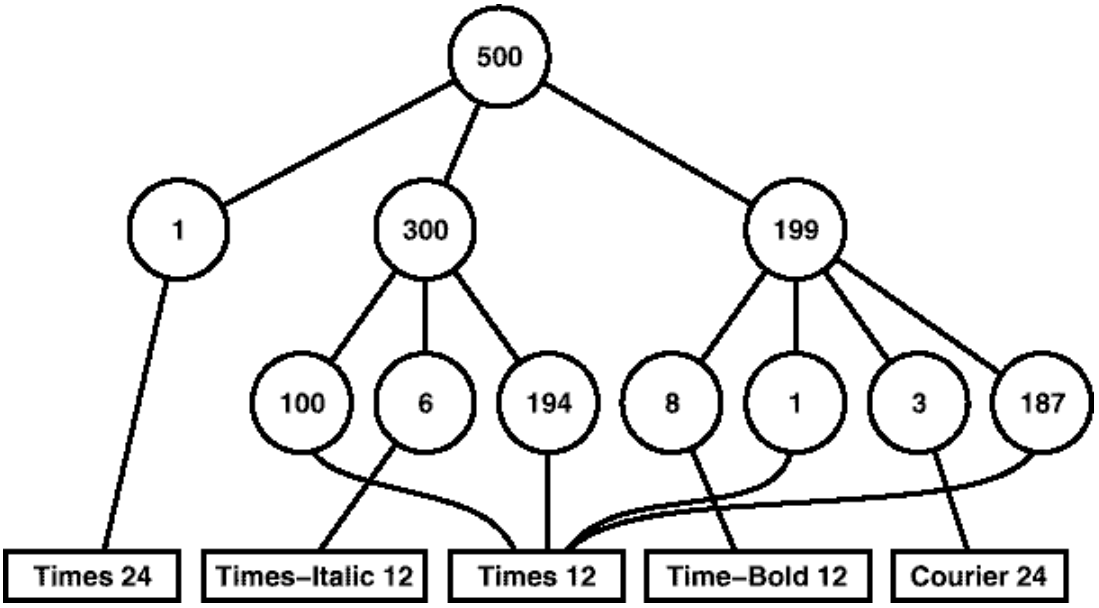
GlyphContext must be kept informed of the current position in the glyph structure during traversal. GlyphContext::Next increments _index as the traversal proceeds. Glyph subclasses that have children (e.g., Row and Column) must implement Next so that it calls Glyph Context::Next at each point in the traversal.

GlyphContext::Get Font uses the index as a key into a BTree structure that stores the glyph-to-font mapping. Each node in the tree is labeled with the length of the string for which it gives font information. Leaves in the tree point toa font, while interior nodes break the string into substrings, one for each child.

Consider the following excerpt from a glyph composition:



The BTree structure for font information might look like

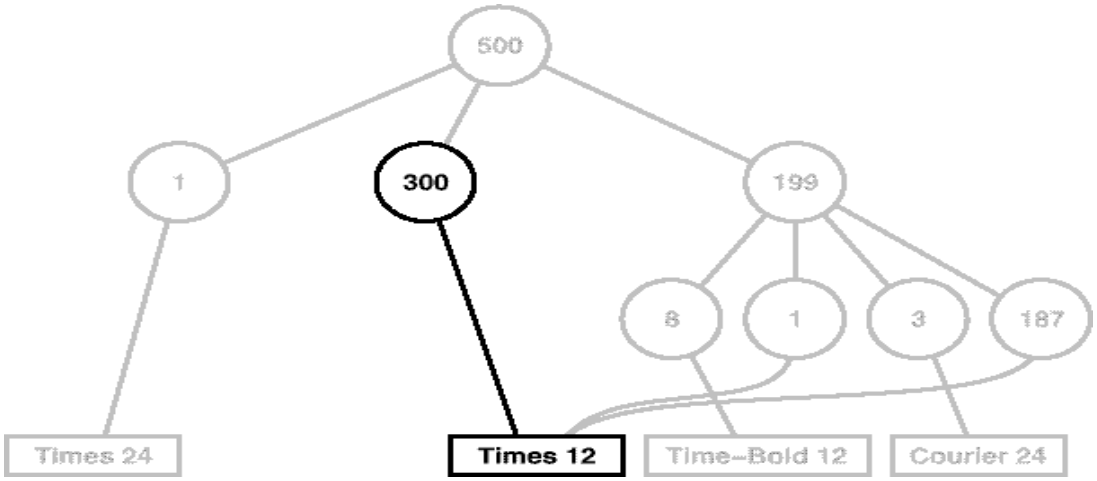


Interior nodes define ranges of glyph indices. BTree is updated in response to font changes and whenever glyphs are added to or removed from the glyph structure. For example, assuming we're at index 102 in the traversal, the following code sets the font of each character in the word "expect" to that of the surrounding text (that is, times12, an instance of Font for 12-point Times Roman):

```
GlyphContext gc;

Font* times12 = new Font("Times-Roman-12");

Font* timesItalic12 = new Font("Times-Italic-12");
```

gc.SetFont(times12, 6);

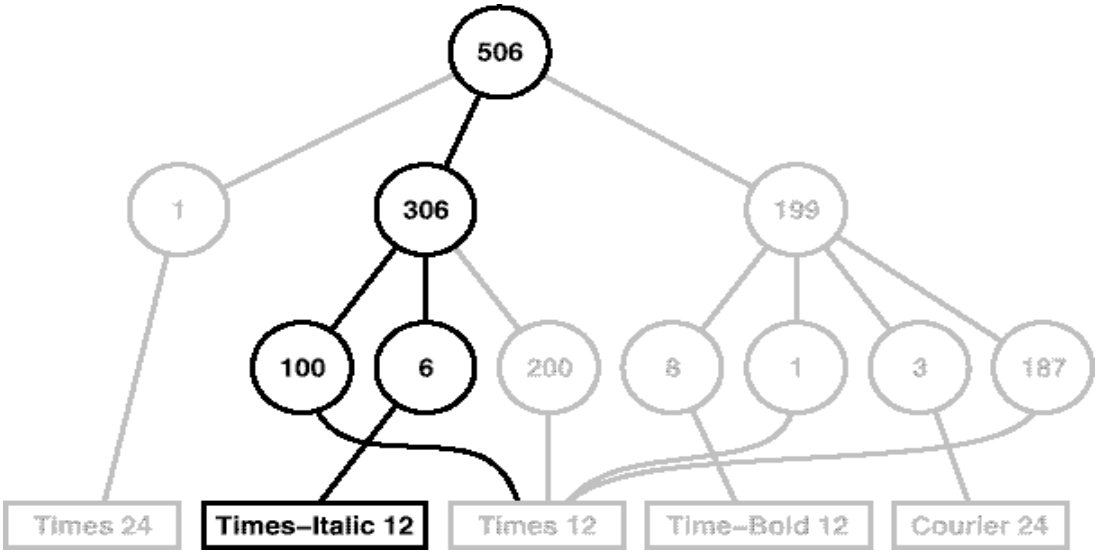
The new BTree structure (with changes shown in black) looks like



Suppose we add the word "don't " (including a trailing space) in 12-point Times Italic before "expect." The following code informs the gc of this event, assuming it is still at index 102:

gc.Insert(6); gc.SetFont(timesItalic12, 6);

The BTree structure becomes



When the GlyphContext is queried for the font of the current glyph, it descends the BTree, adding up indices as it goes until it finds the font for the current index. Because the frequency of font changes is relatively low, the tree stays small relative to the size of

the glyph structure. This keeps storage costs down without an inordinate increase in look-up time.

The last object we need is a FlyweightFactory that creates glyphs and ensures they're sharedproperly. ClassGlyphFactoryinstantiatesCharacterandotherkinds ofglyphs. We only share Character objects; composite glyphs are far less plentiful, and their important state (i.e., their children) is intrinsic anyway.

```
const int NCHARCODES = 128;

class GlyphFactory { public:
    GlyphFactory();
    virtual ~GlyphFactory();




    virtual Character* CreateCharacter(char); virtual Row* CreateRow();
    virtual Column* CreateColumn();

    // ... private:
    Character* _character[NCHARCODES];
};
```

The _character array contains pointers to Character glyphs indexed by character code. The array is initialized to zero in the constructor.

```
GlyphFactory::GlyphFactory () {
    for (int i = 0; i < NCHARCODES; ++i) {
                                _character[i] = 0;
    }
}
```

CreateCharacter looks up a character in the character glyph in the array, and it returns the corresponding glyph if it exists. If it doesn't, then CreateCharacter creates the glyph, puts it in the array, and returns it:

```
Character* GlyphFactory::CreateCharacter (char c) { if (!_character[c]) {
        _character[c] = new Character(c);
    }
    return _character[c];
}
```

The other operations simply instantiate a new object each time they're called, since noncharacter glyphs won't be shared:

```
Row* GlyphFactory::CreateRow () { return new Row;
}

Column* GlyphFactory::CreateColumn () { return new Column;
}
```

We could omit these operations and let clients instantiate unshared glyphs directly. However, if we decide to make these glyphs sharable later, we'll have to change client code that creates them.

Known Uses

- The concept of flyweight objects was first described and explored as a design techniqueinInterViews3.
- Its developers built a powerfuldocument editor called Doc as a proof of concept [CL92]. Doc uses glyph objects to represent each characterinthedocument.
- The editor builds one Glyph instance for each character inaparticularstyle(which defines its graphical attributes); hence a character's intrinsic state consists of the charactercodeanditsstyleinformation(anindex into a style table).[4]
- That means only position is extrinsic, making Doc fast. Documents are represented by a class Document, which also acts as the Flyweight Factory. Measurements on Doc have shown that sharing flyweight characters is quite effective. In a typical case, a document containing 180,000 characters required allocation of only 480 character objects.

- ET++ [WGM88] uses flyweights to support look-and-feel independence.

- The look-and-feelstandardaffectsthelayoutofuserinterfaceelements(e.g.,scroll bars,buttons,menus—knowncollectivelyas"widgets")andtheirdecorations(e.g., shadows, beveling).

Related Patterns

- The Flyweight pattern is often combined with the Composite (183) pattern to implementalogicallyhierarchicalstructureintermsofadirected-acyclicgraph with shared leaf nodes.

- It's often best to implement State and Strategy objects as flyweights.