Software Architecture and Design Pattern

Module III

Design Pattern Catalog: Structural patterns, Adapter, bridge, composite, decorator, facade, flyweight, proxy

Prepared by

Umapathi G R
DEPT OF ISE
ACIT

Describing design pattern (How do we describe design pattern?)

Graphical notations, while important and useful, aren't sufficient. They simply capture the end product of the design process as relationships between classes and objects.

To reuse the design, we must also record the decisions, alternatives, and trade-offs that led to it. Concrete examples are important too, because they help you see the design in action.

We describe design patterns using a consistent format. Each pattern is divided into sections according to the following template. The template lends a uniform structure to the information, making design patterns easier to learn, compare, and use.

The **templates used are:-**

Pattern Name and Classification

The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary.

Intent

A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

Also Known As

Motivation

A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem

Applicability

What are the situations in which the design pattern can be applied?

What are examples of poor designs that the pattern can address? How can you recognize these situations?

Structure

A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT).

Participants

The classes and/or objects participating in the design pattern and their responsibilities.

Collaborations

How the participants collaborate to carry out their responsibilities.

Consequences

How does the pattern support its objectives?

What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?

Implementation

What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?

Sample Code

Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.

Known Uses

Examples of the pattern found in real systems. We include at least two examples from different domains.

Related Patterns

What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

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

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

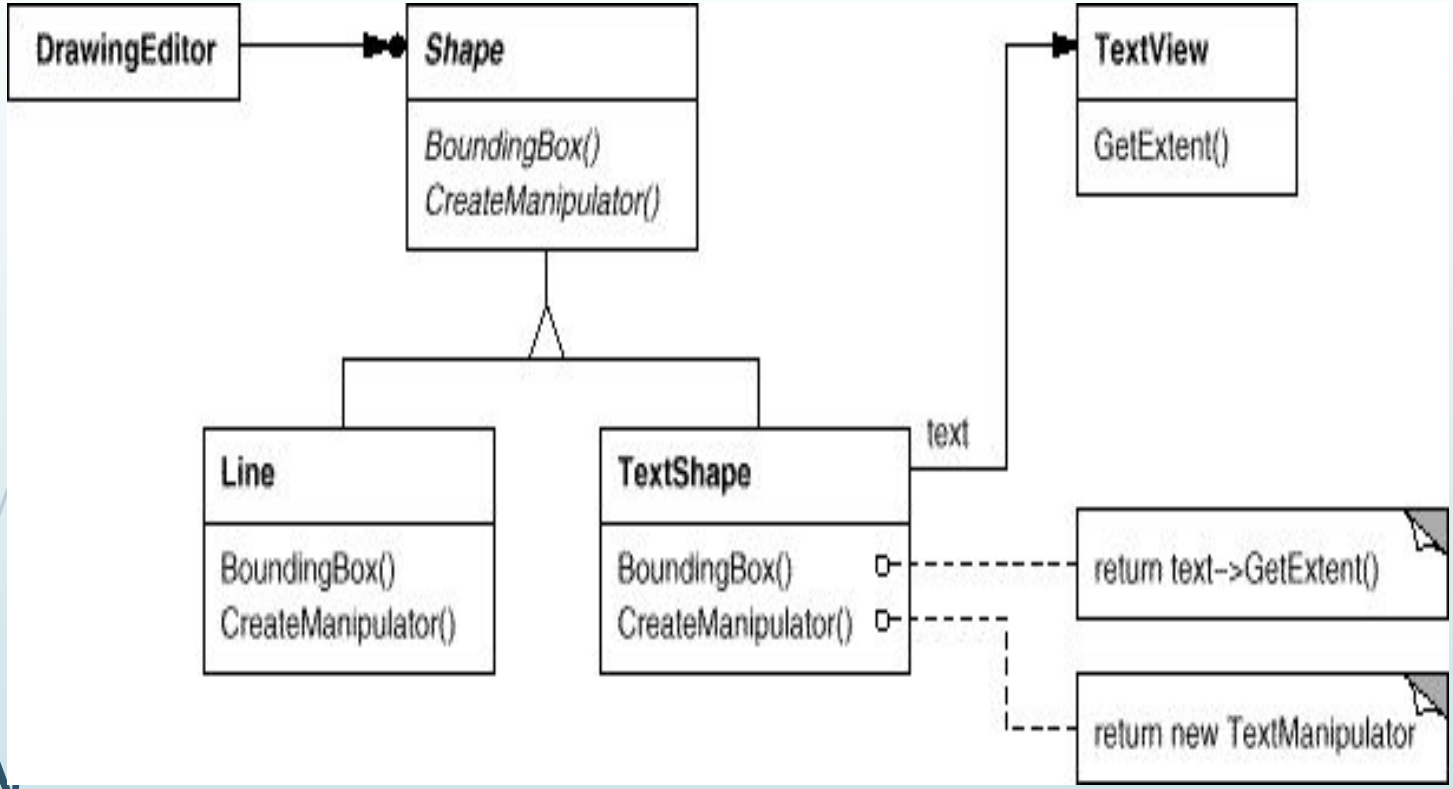
Wrapper

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.


DrawingEditor
Shape
BoundingBox()
CreateManipulator()
TextView
GetExtent()
Line
BoundingBox()
CreateManipulator()
TextShape
BoundingBox()
CreateManipulator()
text
return text->GetExtent()
return new TextManipulator

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

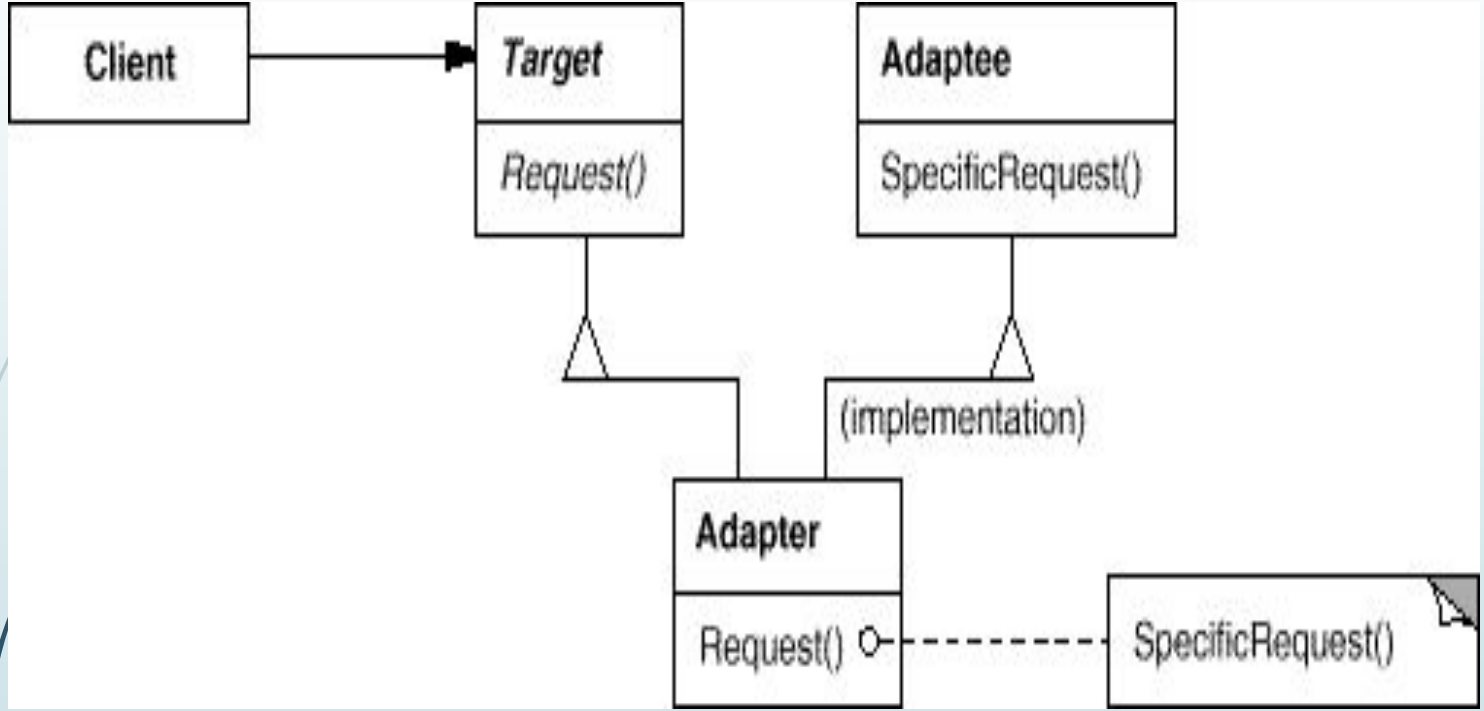
We want to use an existing class, and its interface does not match the one you need.

We want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.

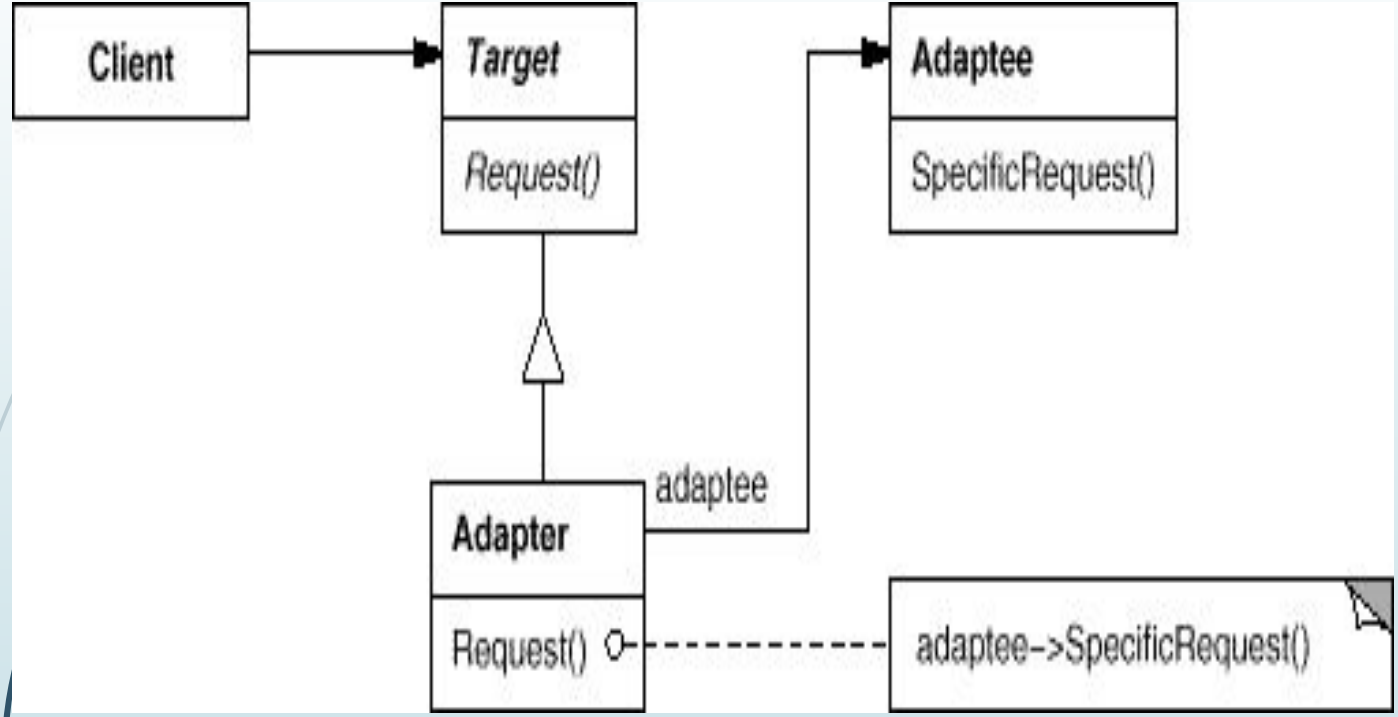
(object adapter only) We need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Structure

class adapter uses multiple inheritance to adapt one interface to another



An object adapter relies on object omposition:



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 nterface

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.

Implementation

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

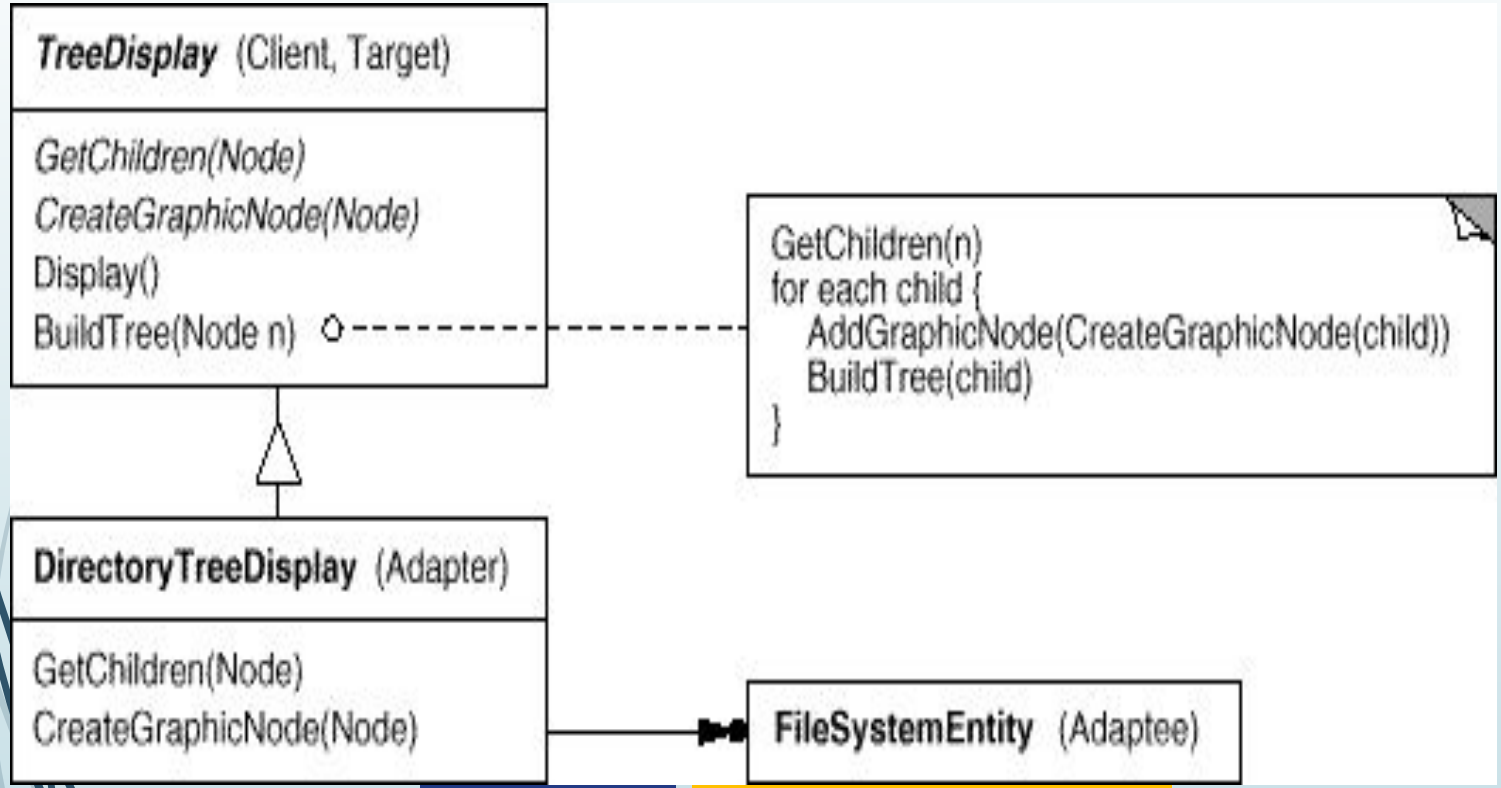
Implementing class adapters in C++.

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.

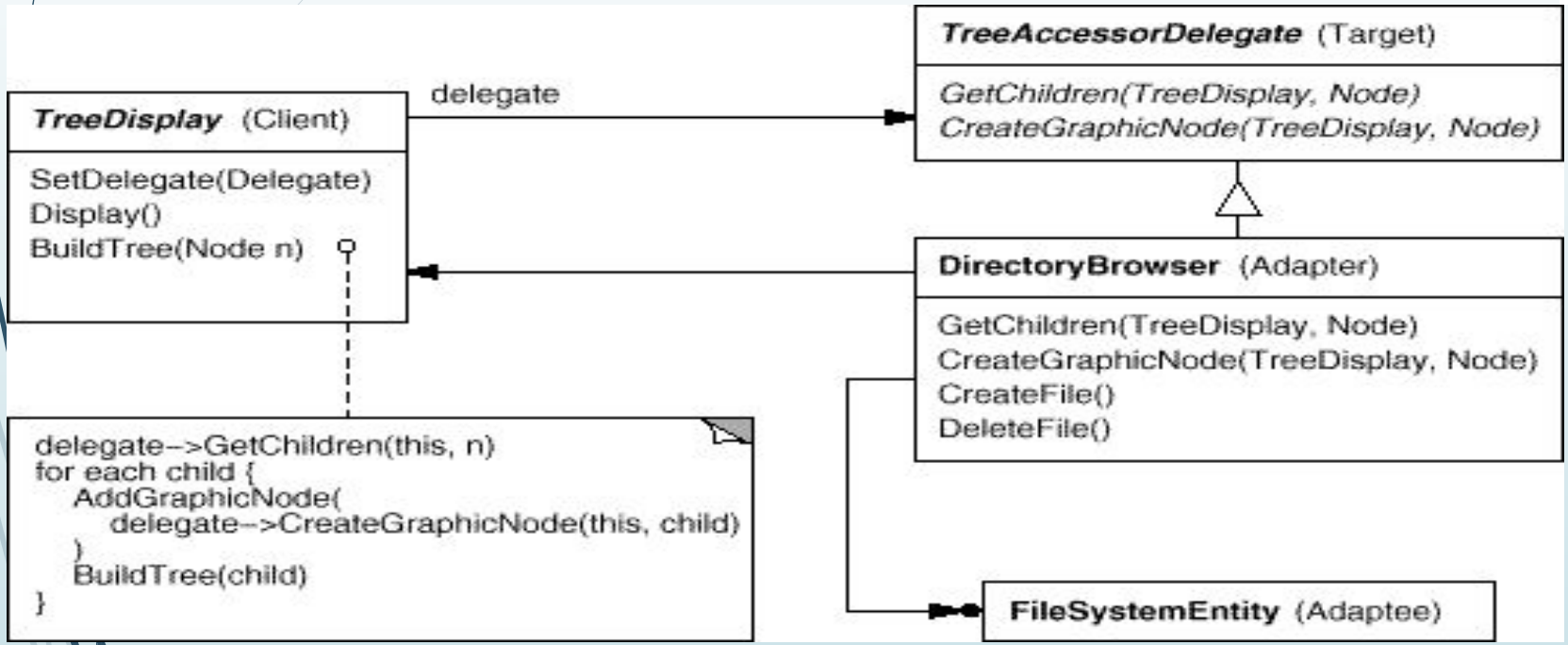
The narrow interface leads to three implementation approaches:

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



Usingdelegateobjects

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.



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

18

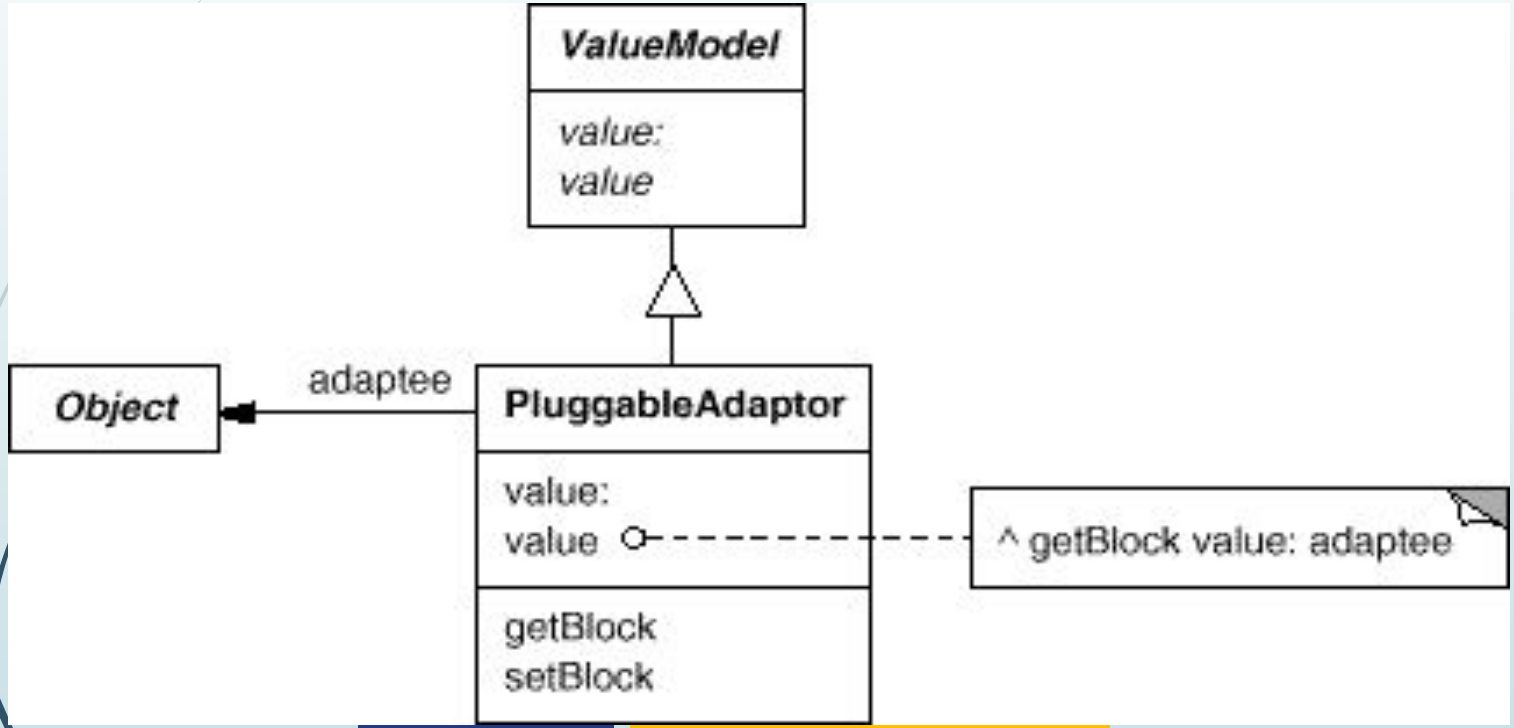
We'll give a brief sketch of the implementation of class and object adapters forthe Motivation example beginning with the classes Shape and TextView.

```
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;
```

Known uses

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

Intent

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

Also Known As Surrogate

Motivation

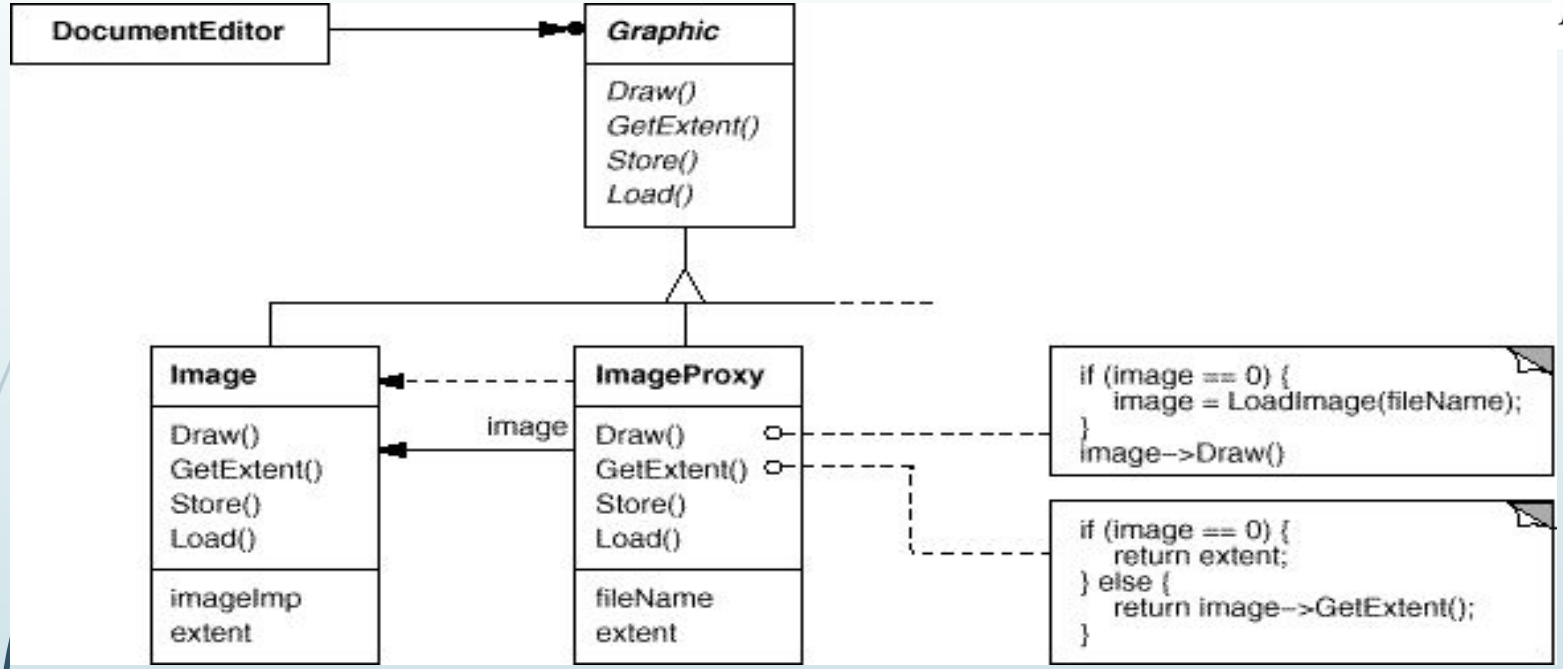
One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it. Consider a document editor that can embed graphical objects in a document. Some graphical objects,like large raster images, can be expensive to create.

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.

Diagram shows the DocumentEditorclass



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.

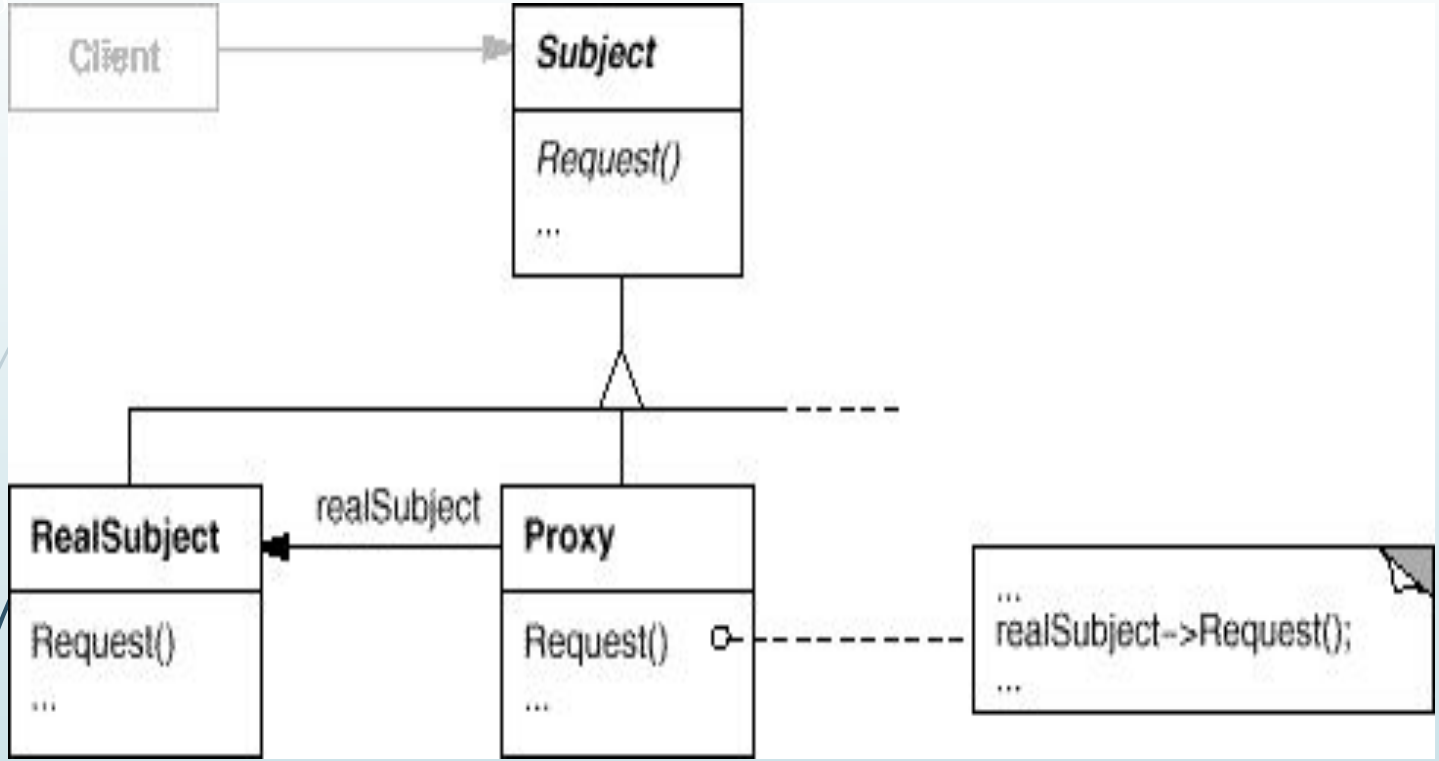
A **virtual proxy** creates expensive objects on demand. The Image Proxy described in the Motivation is an example of such a proxy.

A **protection proxy** controls access to the original object. Protection proxies are useful when objects should have different access rights.

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

Structure


Client
Subject
Request()
...
RealSubject
Request()
...
realSubject
Proxy
Request()
...
...
realSubject->Request();
...

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:

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.

Implementation

The Proxy pattern can exploit the following language features:

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:
```

```
ImpImage* LoadImage(); private:
Image* _image;

const char* _imageFile;

};

ImagePtr::ImagePtr (const char* theImageFile) {

imageFile = theImageFile;

image = 0;

}
```

ImUsing **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.

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.

Known Uses

The virtual proxy example in the Motivation section is from the text buildingblock classes.NEXTSTEP 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.

.

Applicability

Use the Bridge pattern when

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.

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

Participants

Abstraction (Window)

defines the abstraction's interface. maintains a reference to an object of type Implementer.

Refined Abstraction (IconWindow)

Extends the interface defined by Abstraction.

Implementer (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 Implementer 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 Implementer object.

Consequences

The Bridge pattern has the following consequences:

Decoupling interface and implementation.

Decoupling Abstraction and Implementer 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.

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

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).

Known uses

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.

Implementation

Consider the following implementation issues when applying the Bridge pattern:

Only one Implementor. In situations where there's only one implementation, creating an abstract Implementor class isn't necessary

creating the right Implementor object.

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

Using multiple inheritance. You can use multiple inheritance in C++ to combine an interface with its implementation

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

Decorator

Intent

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

Also Known As

Wrapper

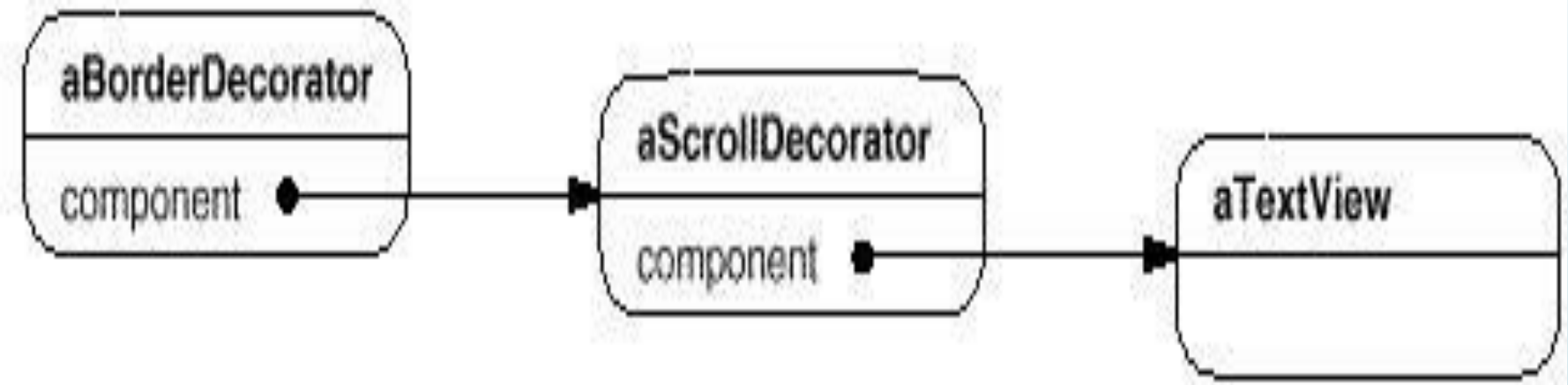
Motivation

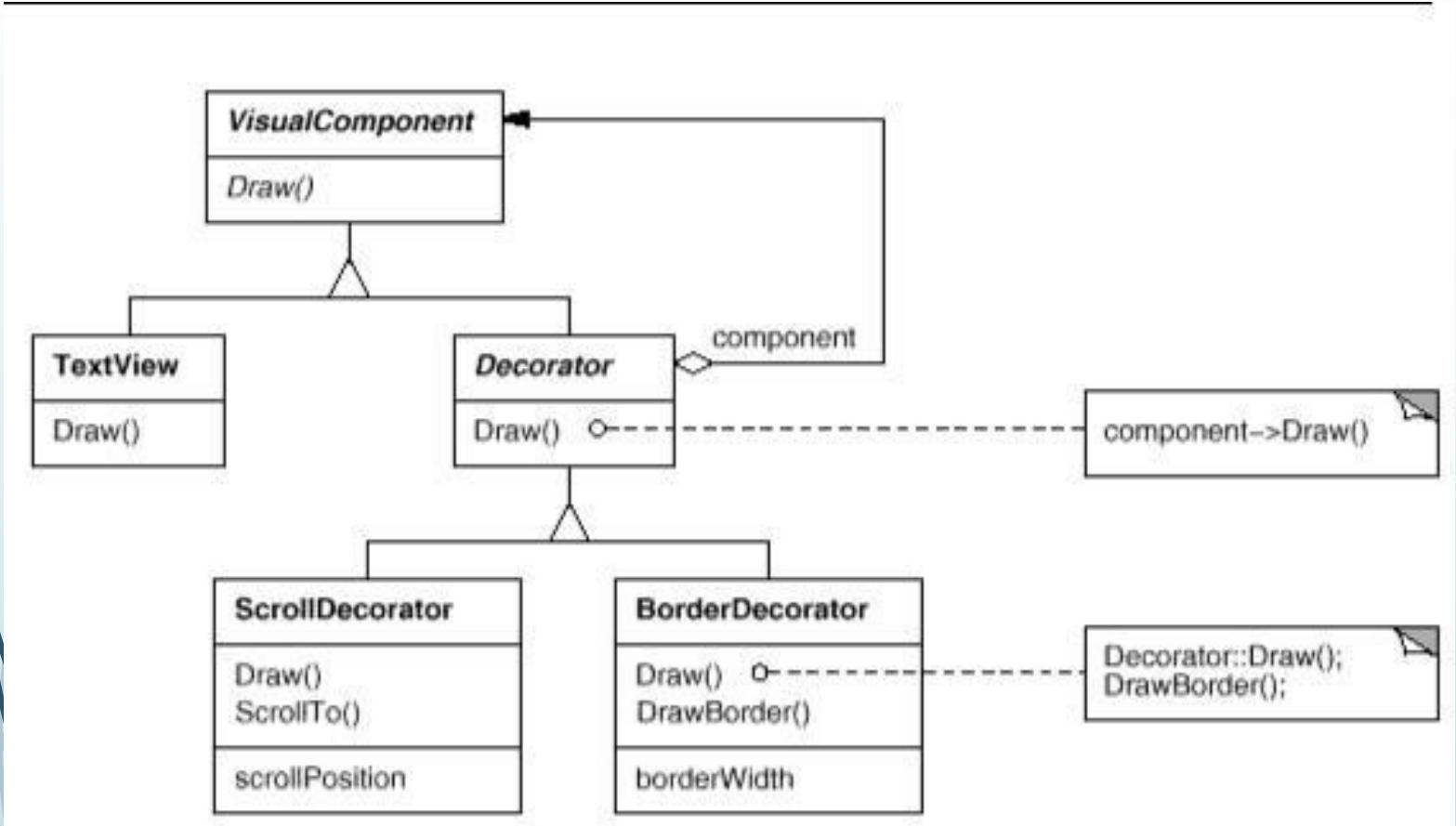
A more flexible approach is to enclose the component in another object that adds the border. The enclosing object is called a decorator.

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.

The following object diagram shows how to compose a TextView object with Borde rDecorator and Scroll Decorator 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
Draw()
TextView
Draw()
Decorator
Draw()
component
component->Draw()
ScrollDecorator
Draw()
ScrollTo()
scrollPosition
BorderDecorator
Draw()
DrawBorder()
borderWidth
Decorator::Draw();
DrawBorder();

Visual Component 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 Scroll To operation lets other objects scroll the interface if they know there happens to be a ScrollDecorator object in the interface.

APPLICABILITY

Use Decorator

45

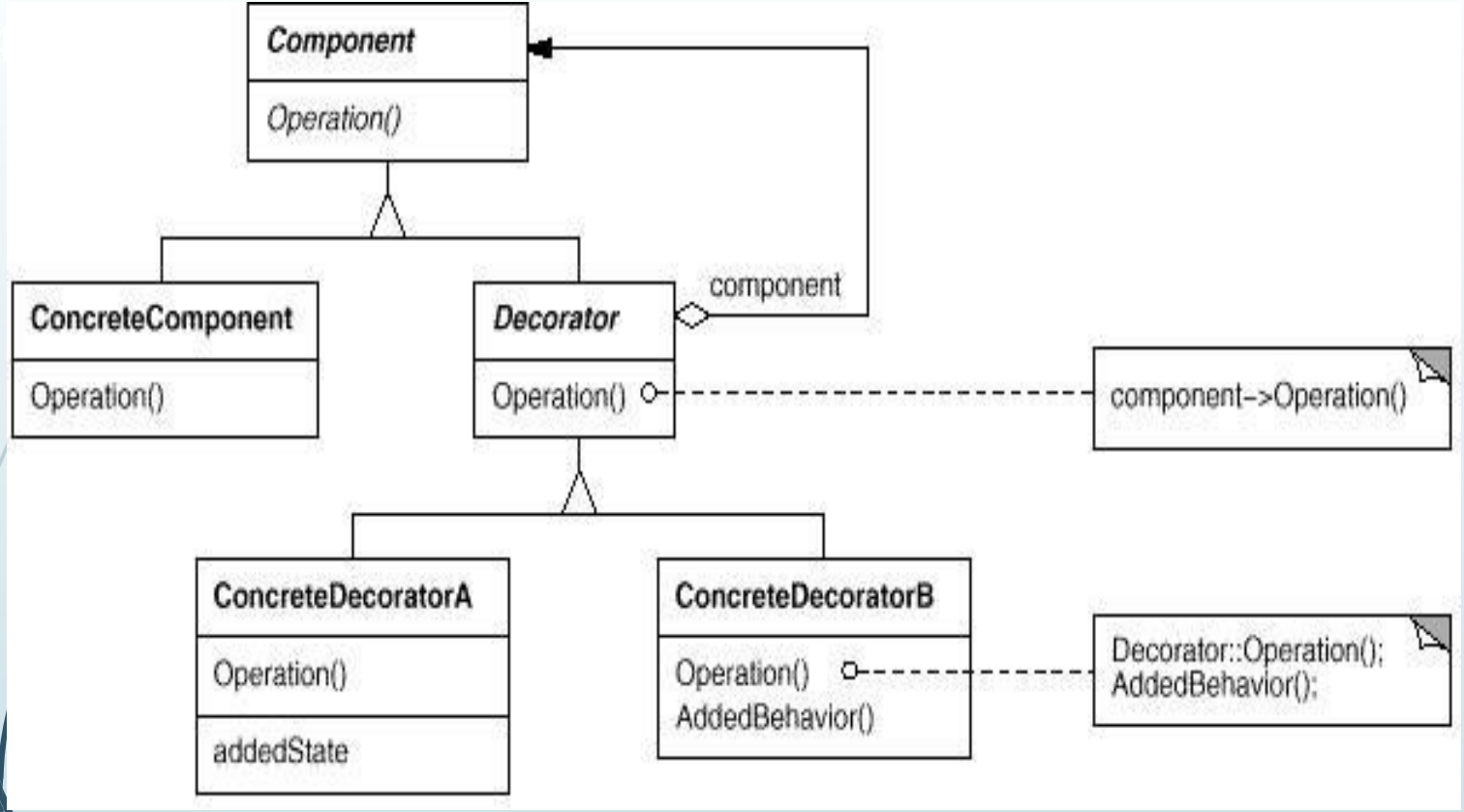
To add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.

for responsibilities that can be withdrawn.

when extension by sub classing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for subclassing.

Structure

.


Component
Operation()
ConcreteComponent
Operation()
Decorator
Operation()
component
component->Operation()
ConcreteDecoratorA
Operation()
addedState
ConcreteDecoratorB
Operation()
AddedBehavior()
Decorator::Operation();
AddedBehavior();

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 thatconforms 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:

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.

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

A decorator and its component aren't identical. A ecorator acts as a transparent enclosure

Lots of little objects. A design that uses Decorator often results in systems composed of lots of little objects that all look alike.

.

.

Implementation

49

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. ConcreteDecorator classes must therefore inherit from a common class (at least in C++).

Omitting the abstract Decorator class. There's no need to define an abstract Decorator class when you only need to add one responsibility

hanging 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.

.

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

.

.

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 Composite ():

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 : 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.

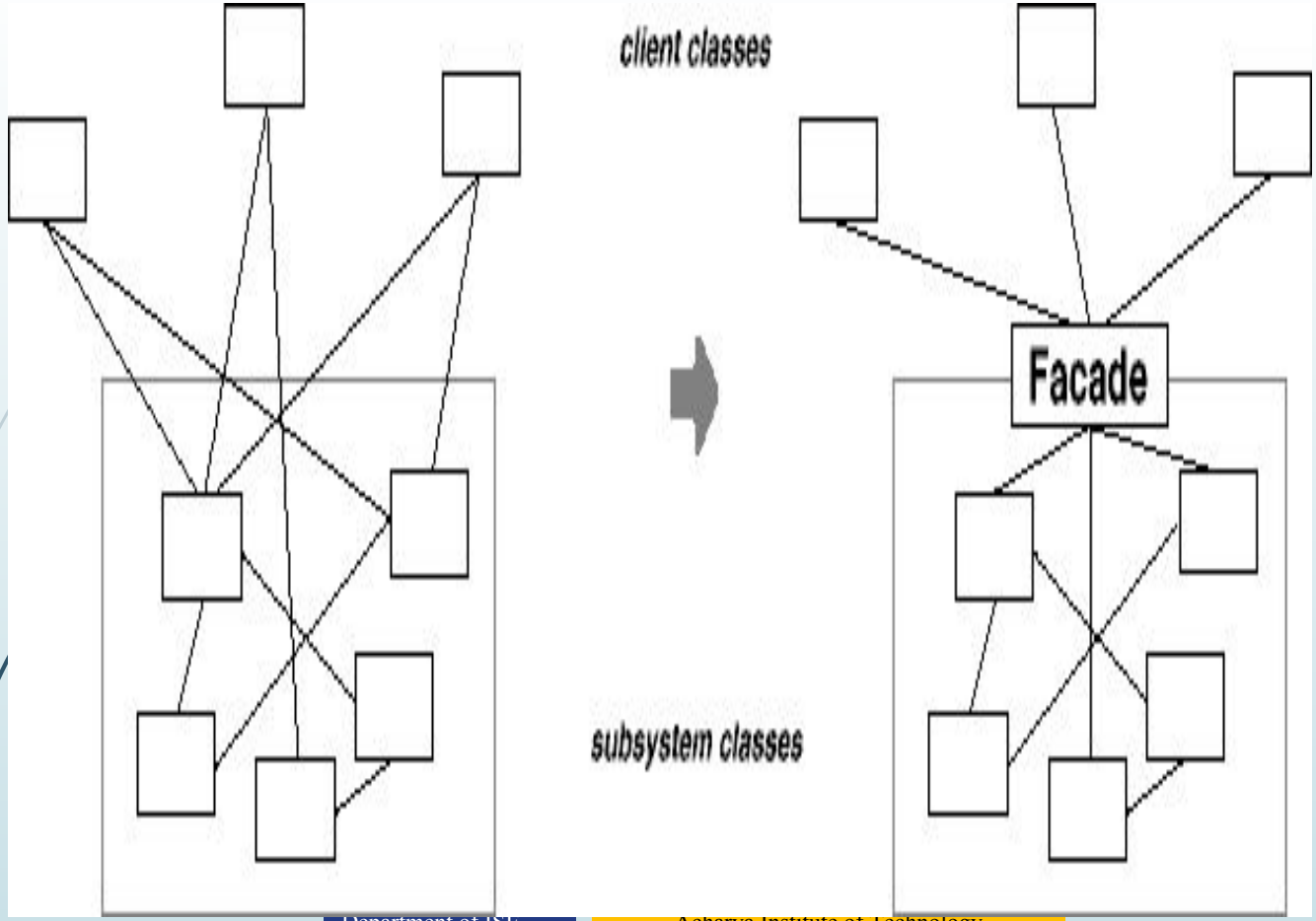
Motivation

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

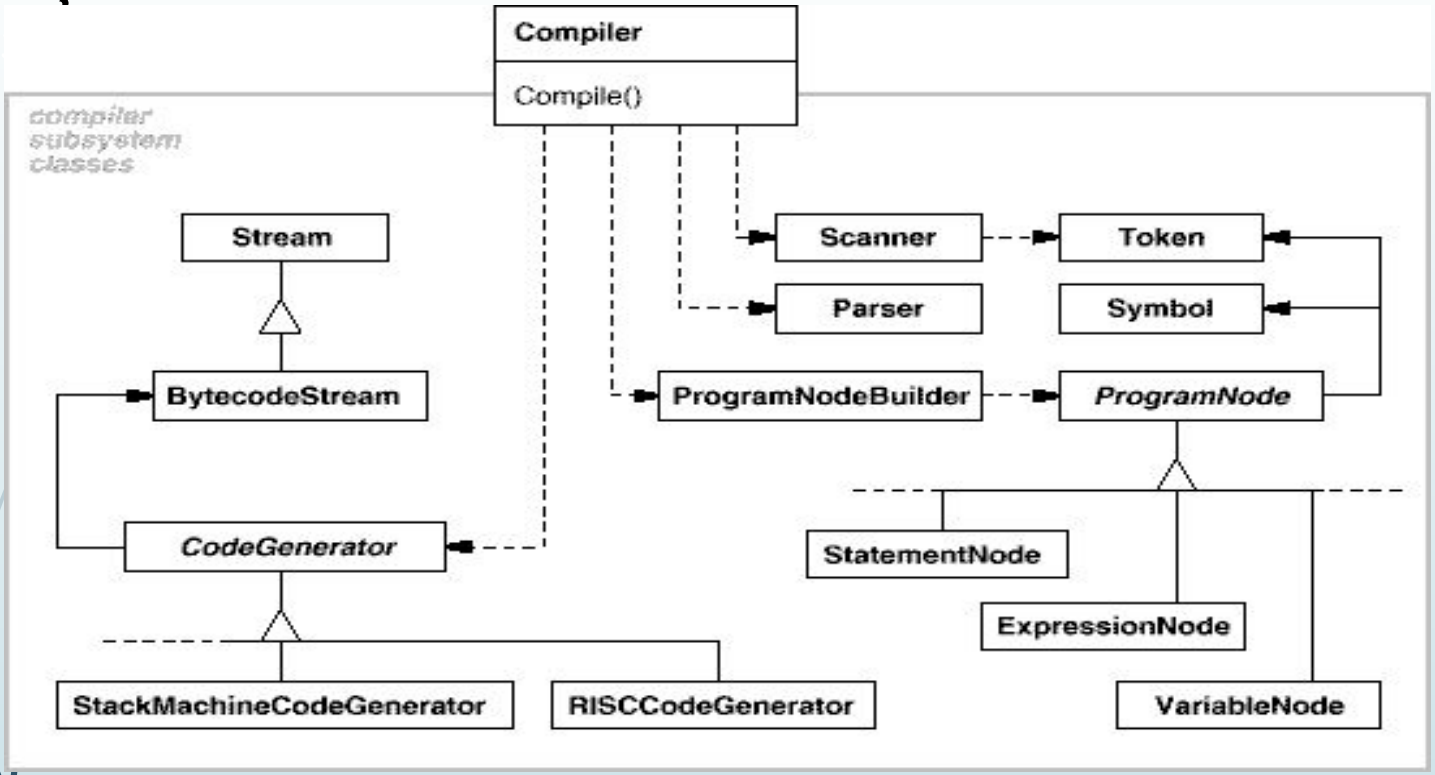
One way to achieve this goal is to introduce a **facade** object that provides a single, simplified interface to the more general facilities of a subsystem.

.

.


client classes
Facade
subsystem classes

Façade


Compiler
Compile()
compiler subsystem classes
Stream
Scanner
Token
Parser
Symbol
BytecodeStream
ProgramNodeBuilder
ProgramNode
CodeGenerator
StatementNode
ExpressionNode
StackMachineCodeGenerator
RISCCodeGenerator
VariableNode

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,

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.

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 subsystem are strongly coupled. Weak coupling lets you vary the components of the subsystemwithoutaffectingitsclients.

Facades help layer a system and the dependencies between objects. They can eliminate complex orcircular dependencies.Thiscanbeanimportan tconsequence 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.

.

Public versus private subsystem classes.

A subsystem is analogous to a class in that both have interfaces, and both encapsulate something—a class encapsulates state and operations, while a subsystem encapsulatesclasses.

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.

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 for different implementations of a 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. .

.

Known Uses

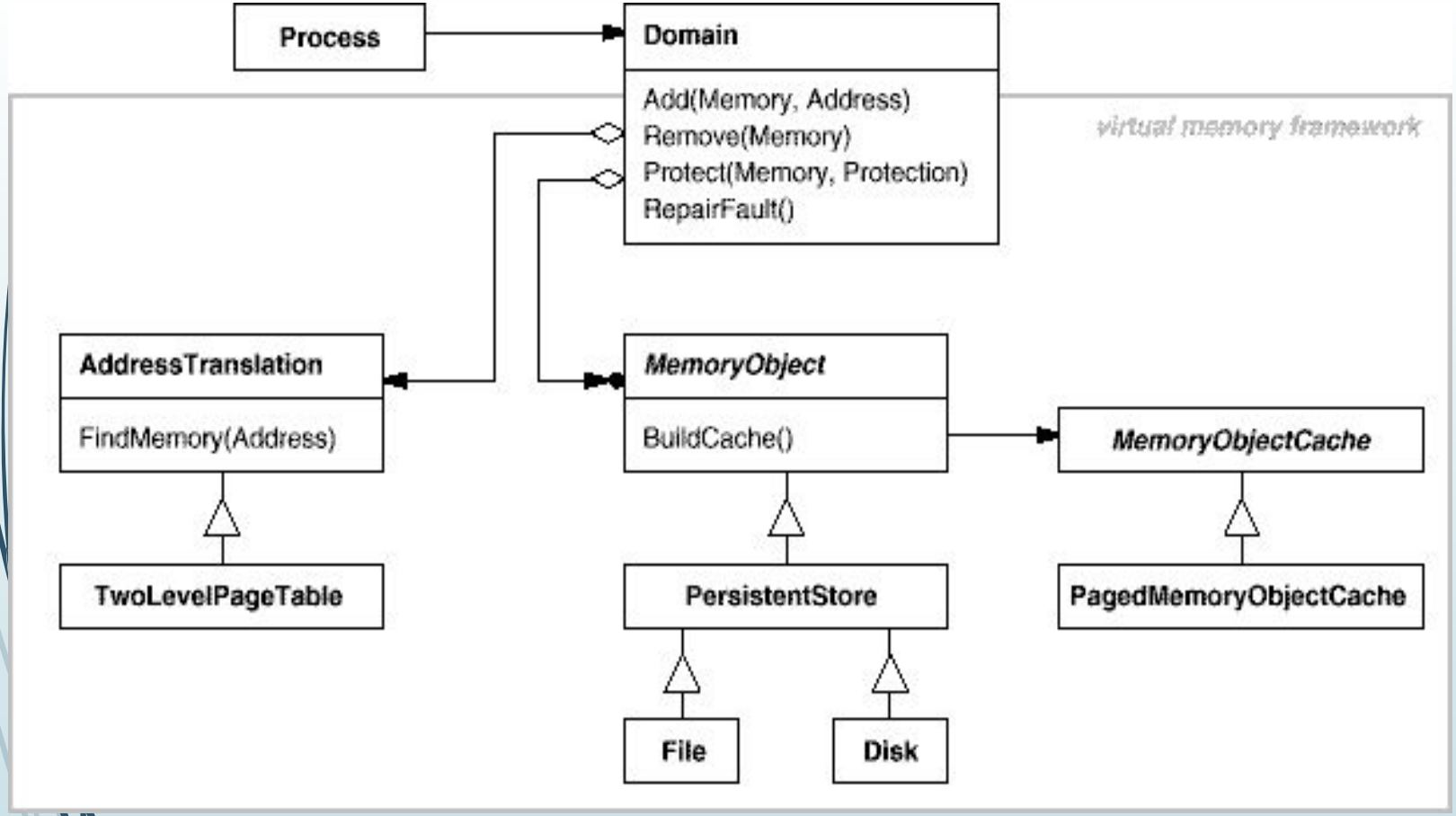
The compiler example in the Sample Code section was inspired by the ObjectWorks\Smalltalk compiler system [Par90].

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..


Process
Domain
Add(Memory, Address)
Remove(Memory)
Protect(Memory, Protection)
RepairFault()
virtual memory framework
AddressTranslation
FindMemory(Address)
MemoryObject
BuildCache()
MemoryObjectCache
TwoLevelPageTable
PersistentStore
PagedMemoryObjectCache
File
Disk

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 memory objects, files, or backing store.

The main operations on Domain support adding a memory object at a particular address, removing a memory object, 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 actuallya 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 Facadein 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

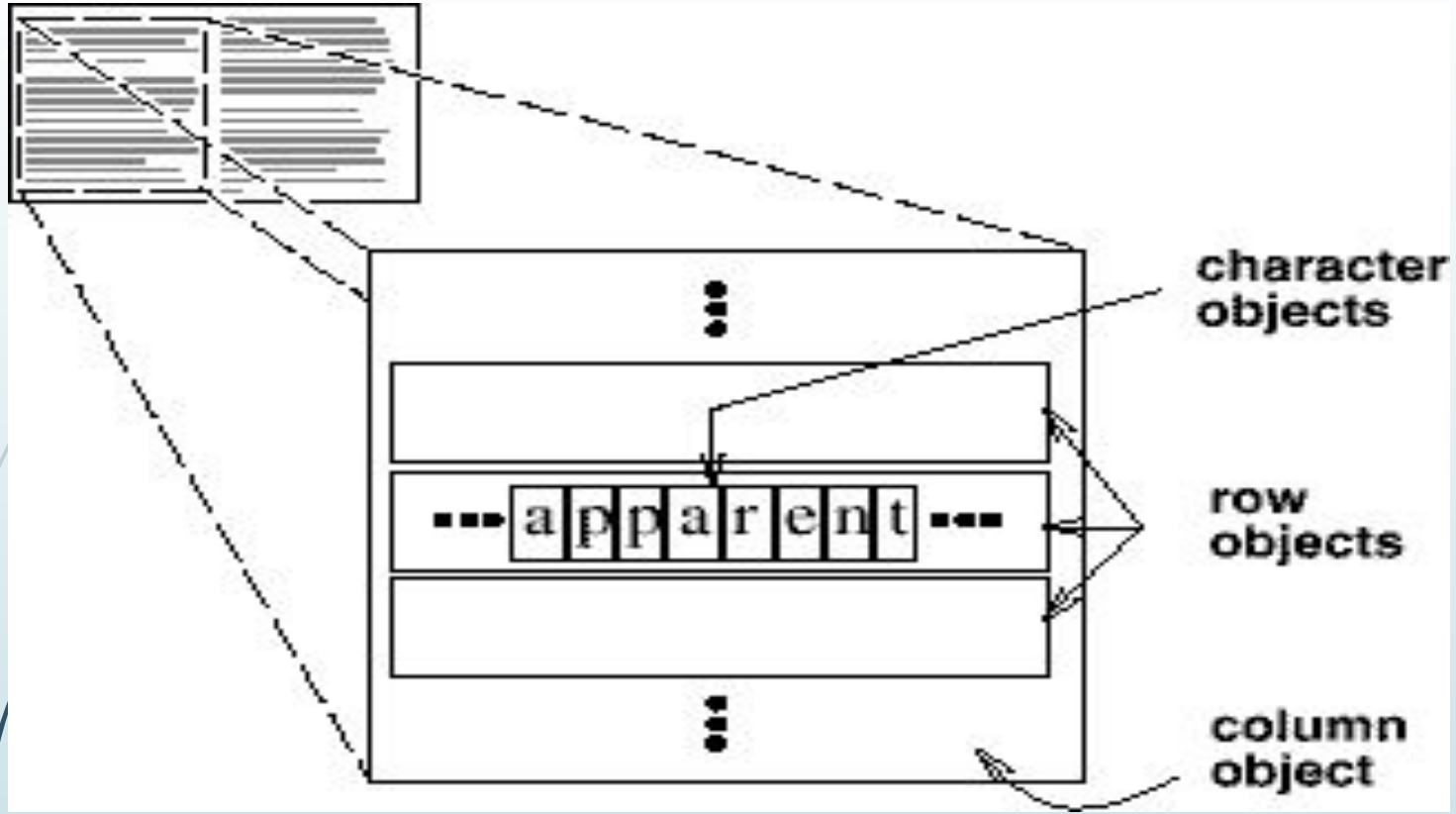
Use sharing to support large numbers of fine-grained objects efficiently.

Motivation

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. However, they usually stop short of using an object for each character in the 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. .


character objects
row objects
column object
a p p a r e n t

The draw backof such 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 pattern describes how to share objects to allow their use at fine granularities without prohibitive cost.

Aflyweight is a shared object that can be used in multiple context simultaneously.

The flyweight acts as an independent object in each context—it's in distinguishable 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.**

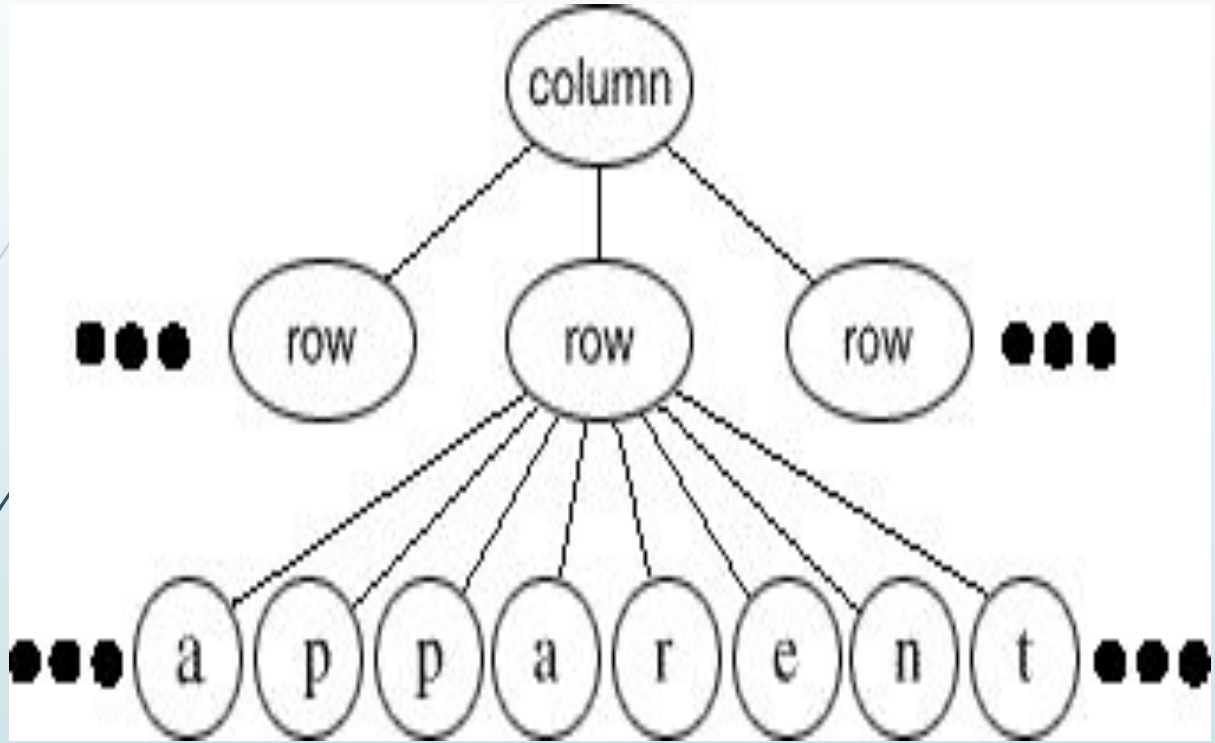
 Intrinsic state is stored in the flyweight; it consists of information that's independent of the flyweight's context, thereby making it sharable.

 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.

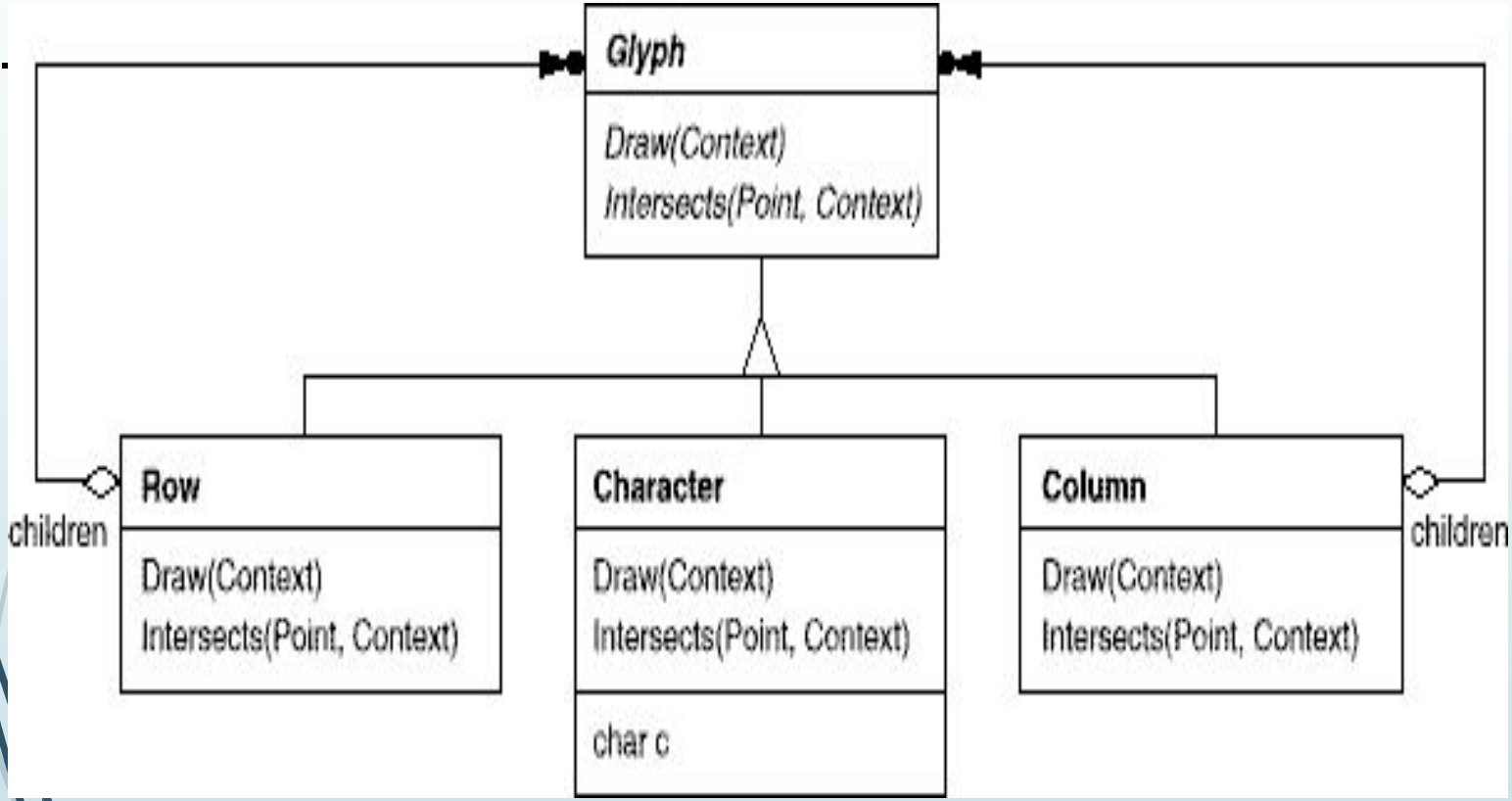
Each flyweight stores a character code, but its coordinate position in the document and its typographic style can be determined from the text layout algorithms and formatting commands in effect wherever the character appears. The character code is intrinsic state, while the other information is extrinsic.

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.


Glyph
Draw(Context)
Intersects(Point, Context)
children
Row
Draw(Context)
Intersects(Point, Context)
Character
Draw(Context)
Intersects(Point, Context)
char c
Column
Draw(Context)
Intersects(Point, Context)
children

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 information that the flyweight needs to draw itself.

For example, 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.

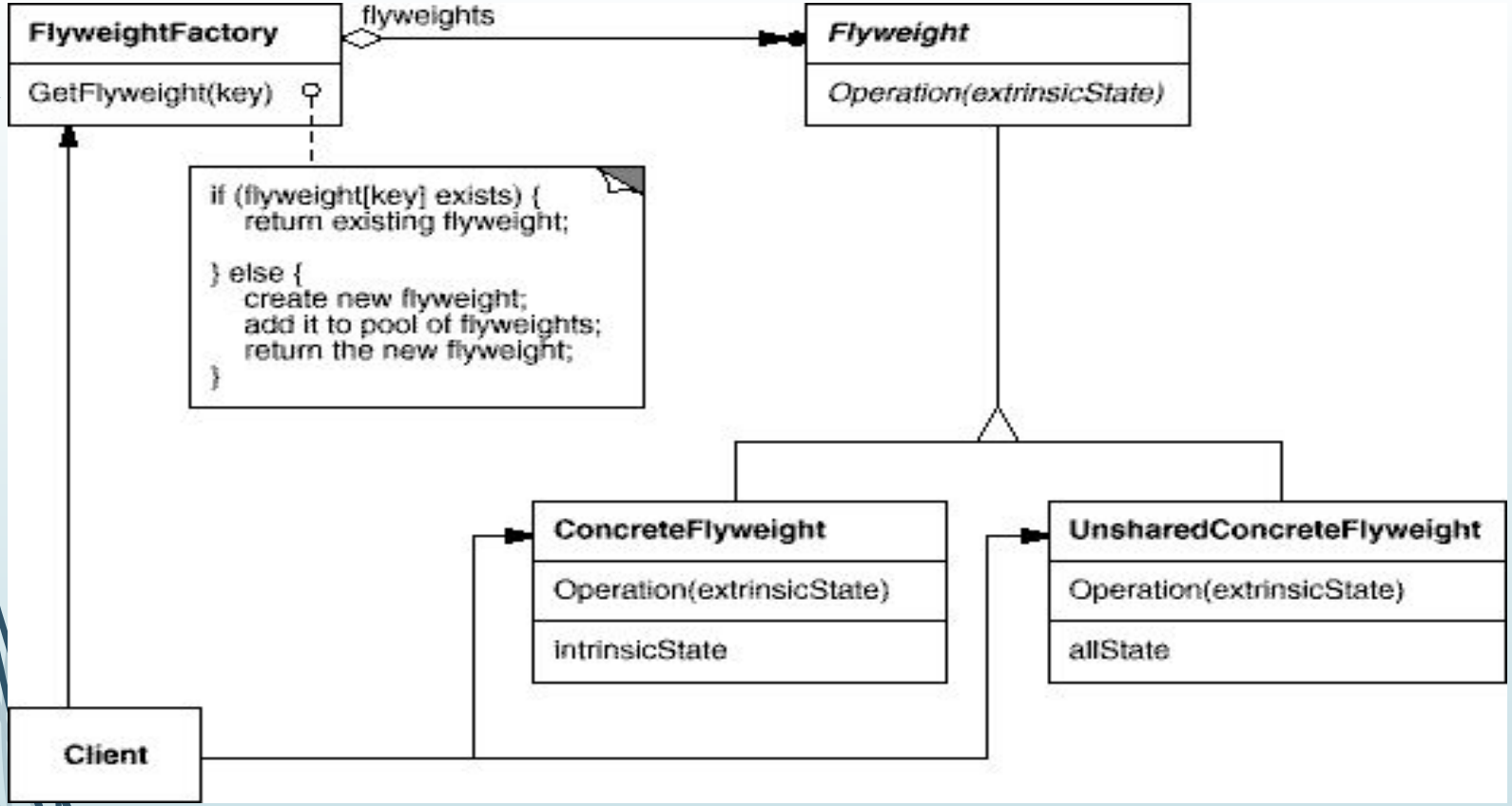
Applicability

The Flyweight pattern's effectiveness depends heavilyon how and where it's used.

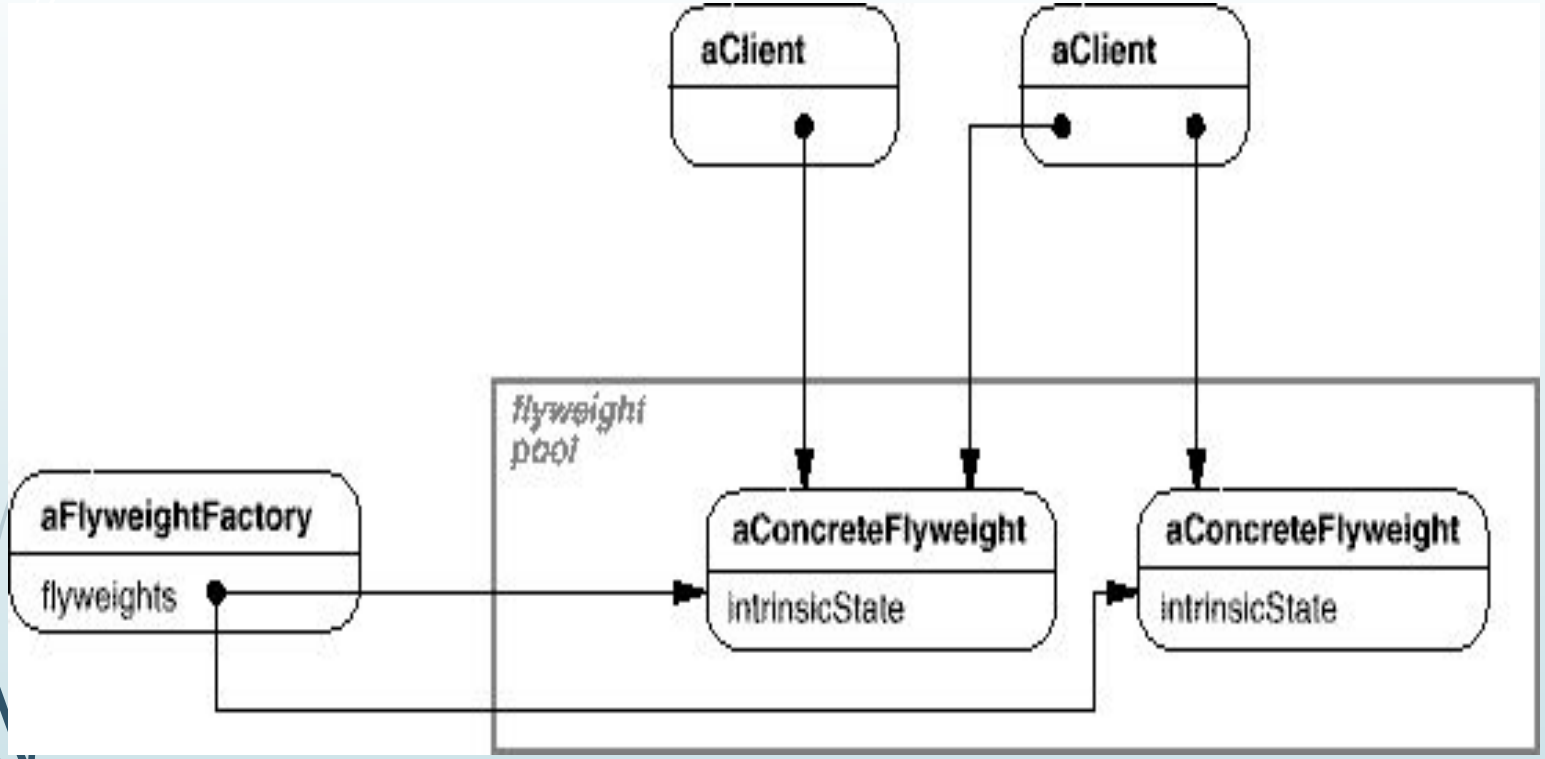
Apply the Flyweight pattern when *all of the following are true:*

- 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.
- The application doesn't depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects

Structure


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

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 must be sharable. 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 Unshared ConcreteFlyweight objects to have ConcreteFlyweight objects as children at some levelin the flyweight object structure (as the Row and Column classes have).

FlyweightFactory

creates and manages flyweight objects.

Ensures that flyweights are shared properly. When a client requests a

flyweight, the Flyweight Factory 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

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.

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 if there are as many different kinds of extrinsic state as the rear eobjects before sharing.

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

Known Uses

The concept of flyweight objects was first described and explored as a design techniqueinInterViews3.

Its developers built a powerful document editor called Doc as a proof of concept [CL92]. Doc uses glyph objects to represent each characte rin the document.

The editor builds one Glyph instance for each character in a particula rstyle(which defines its graphical attributes); hence a character's intrinsic state consists of the character code an dit sstyle in formation(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.

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.

Composite

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

Motivation

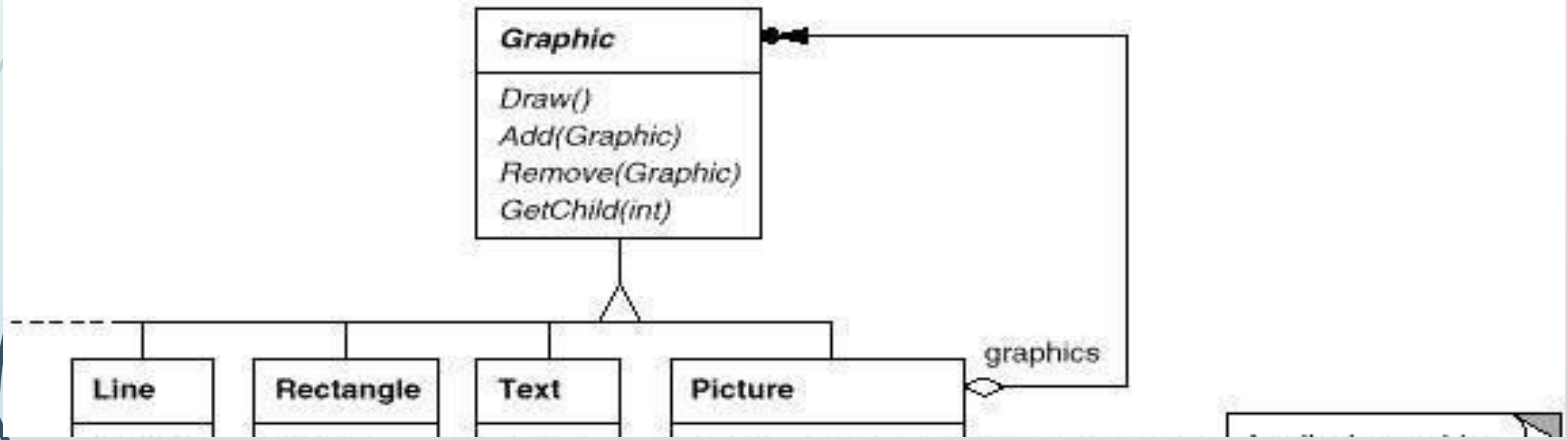
Graphics applications like drawing editors and schematic capture systems let users build complex diagrams out of simple components.

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

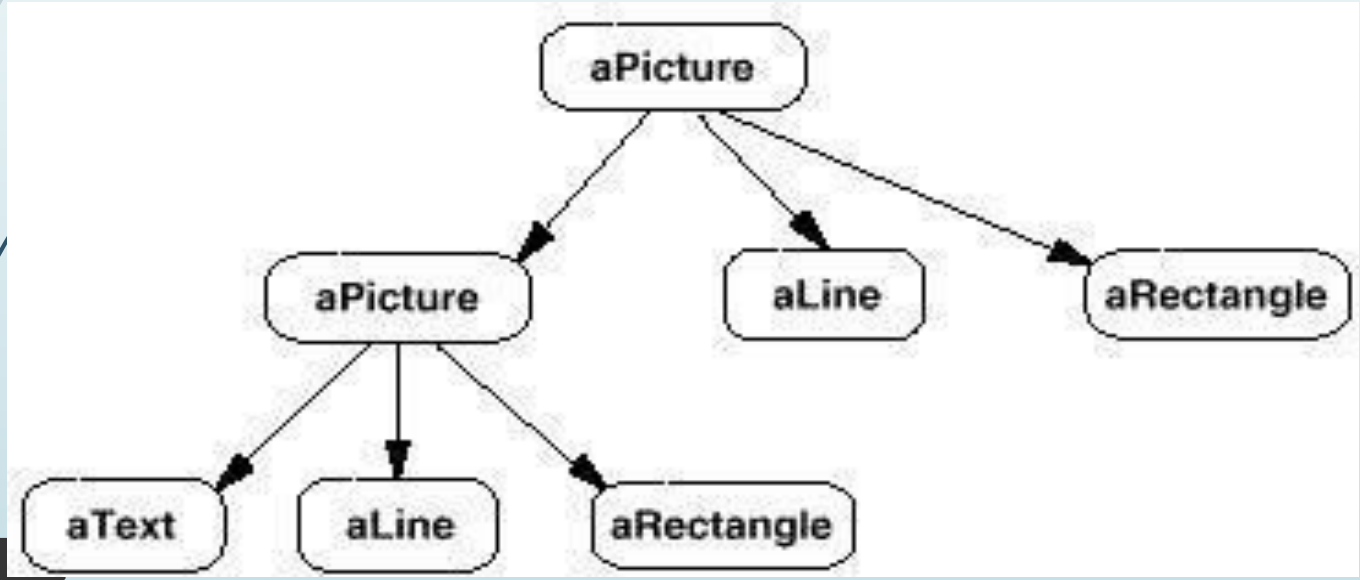
Having to distinguish these objects makes the application more complex. The key to the Composite pattern is an abstract class that represents both primitives and their containers. For the graphics system, this class is Graphic. Graphic declares operations like Draw that are specific to graphical objects.

It also declares operations that all composite objects share, such as operations for accessing and managing its children.



The 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

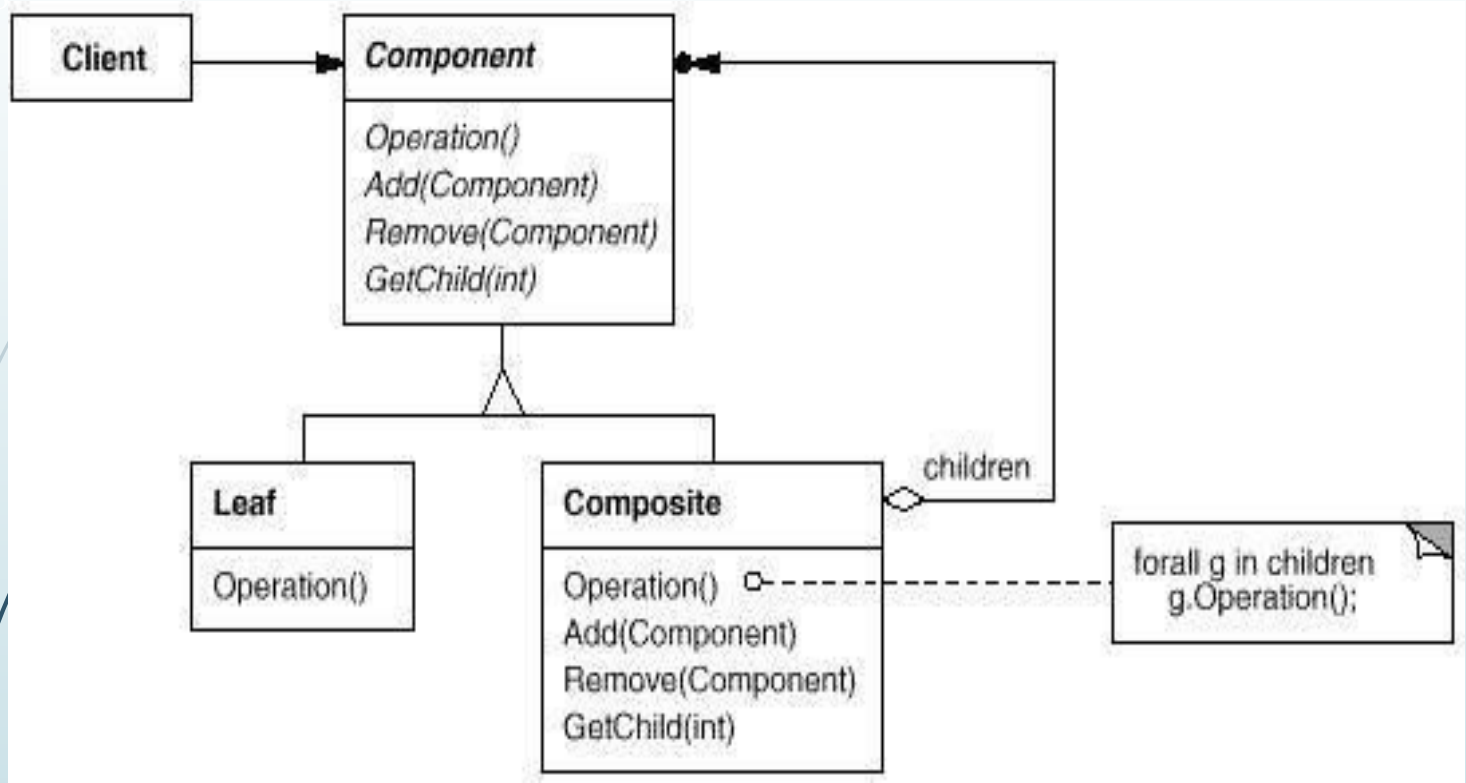
87

Use the Composite pattern when

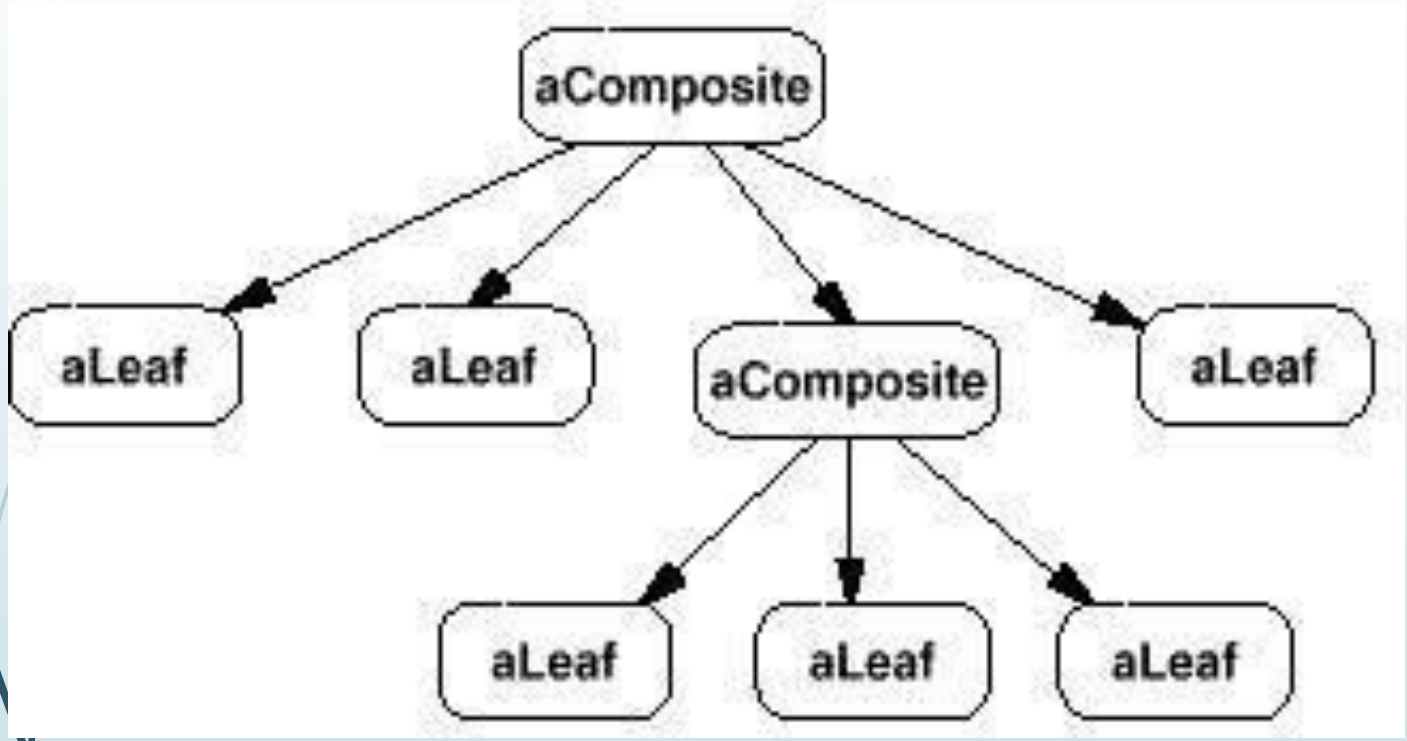
1. To represent part-whole hierarchies of objects.

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


Client
Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)
children
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
forall g in children
g.Operation();

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

92

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

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. .

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. .

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.

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

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