

Design Patterns - Structural

Structural Patterns

Structural Patterns

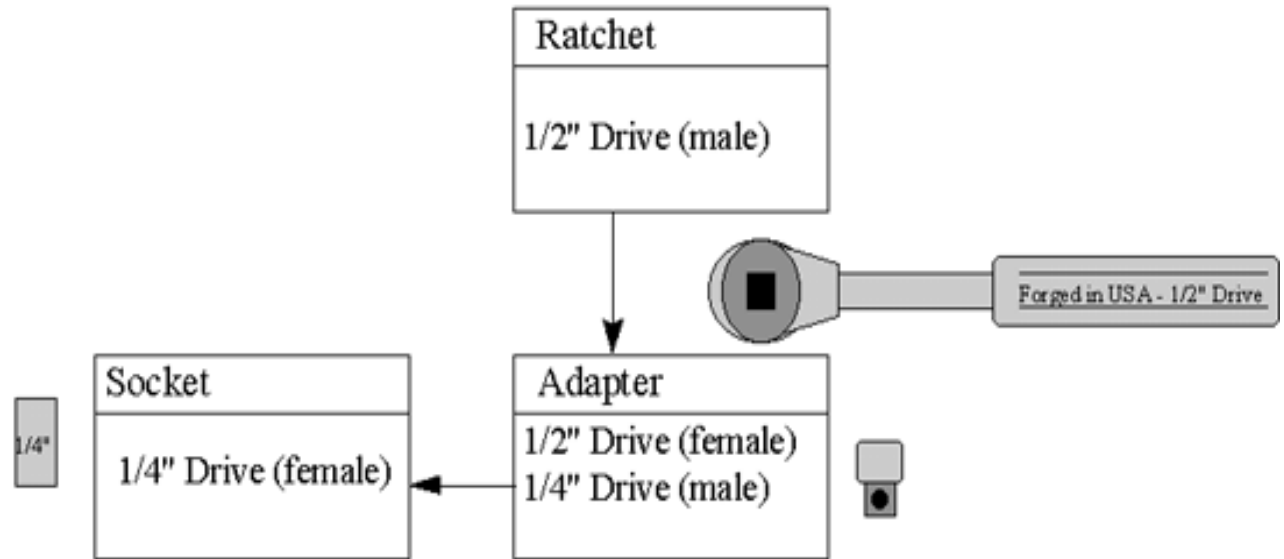
- ▶ Deal with composition of classes or object

•Adapter	•Decorator
•Composite	•Façade
•Proxy	•Flyweight
•Bridge	

Adapter

Adapter

Object Diagram for
Adapter using Socket
Adapter Example



The *Adapter* pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients. Socket wrenches provide an example of the *Adapter*. A socket attaches to a ratchet, provided that the size of the drive is the same. Typical drive sizes in the United States are 1/2" and 1/4". Obviously a 1/2" drive ratchet will not fit into a 1/4" drive socket unless an adapter is used. A 1/2" to 1/4" adapter has a 1/2" female connection to fit on the 1/2" drive ratchet, and a 1/4" male connection to fit in the 1/4" drive socket.

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.

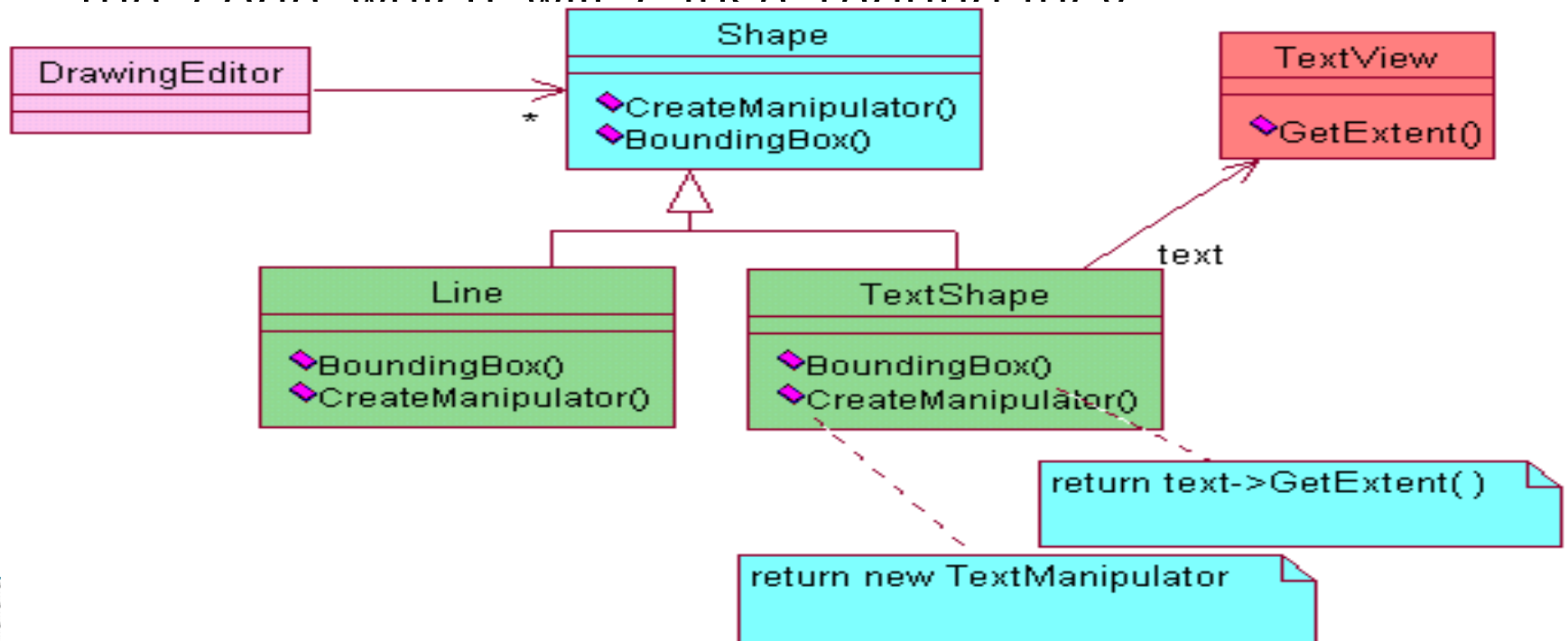
Alternative Name:

Wrapper Problem

- An "off the shelf" component offers compelling functionality that you would like reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed

Illustration: Problem

The classes in the tool kit (Text View in figure) may not be exactly what the client needs, so it becomes difficult for the client to reuse such a class directly. Due to this difficulty one may be tempted to duplicate the code which will cause redundancy.



Solution:

One way to solve this problem is to make the client class (**TextShape**) to adapt the Tool Kit Class (**TextView**) and modify the interface to suit there application. This can be done in two ways

- 1> By additionally inheriting the **TextView** Class or
- 2> Keeping a reference of the tool kit class inside the **Test Shape Class**.

The second method is preferred over the first method as the client hierarchy is kept low in depth, also we are aware of the problems of multiple inheritance.

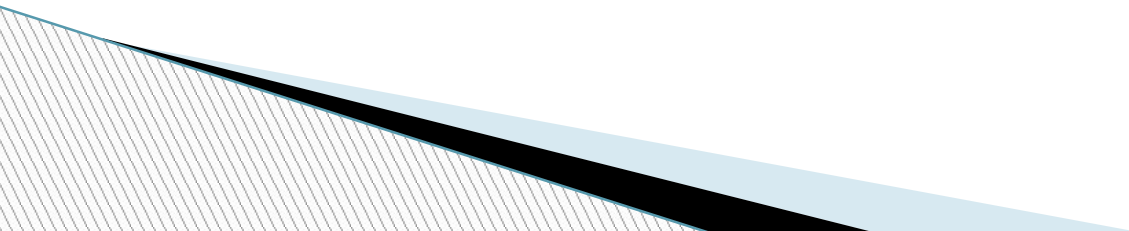
Applicability

- ▶ **Use the Adapter pattern when**

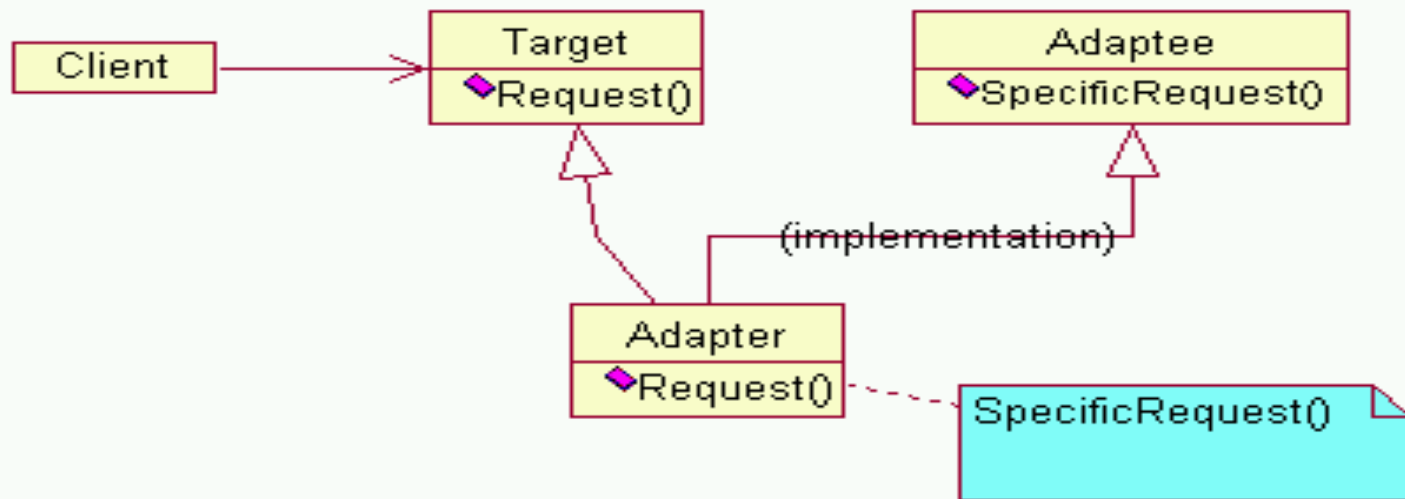
You want to use an existing class, and its interface does not match the one you need.

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

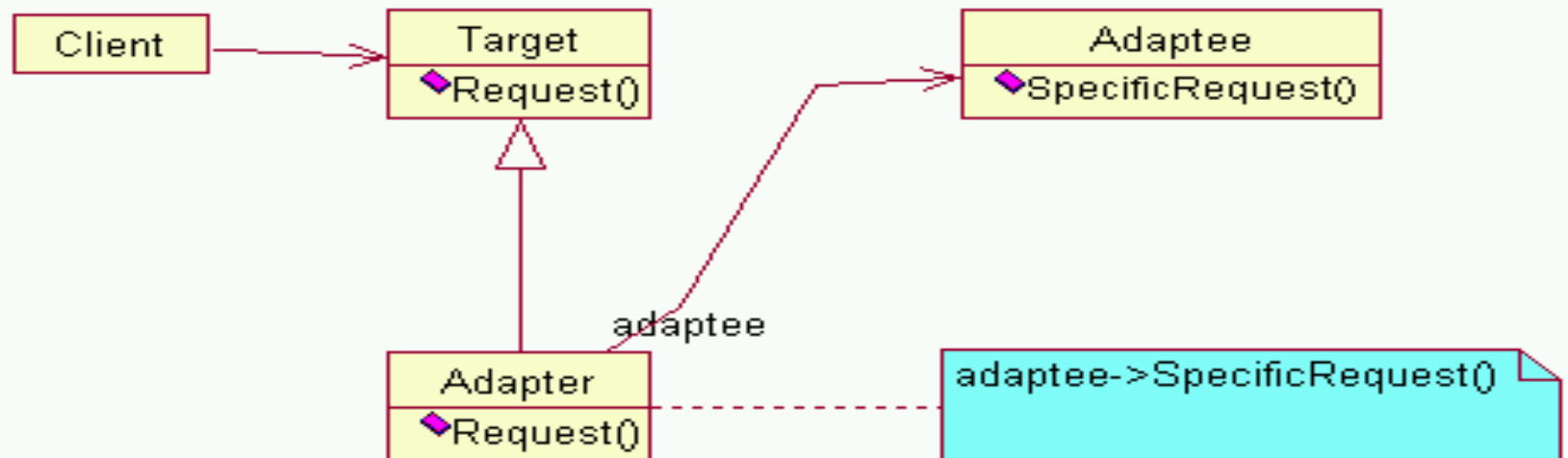
(Object adapter only) you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.



Structure



An object adapter relies on object composition



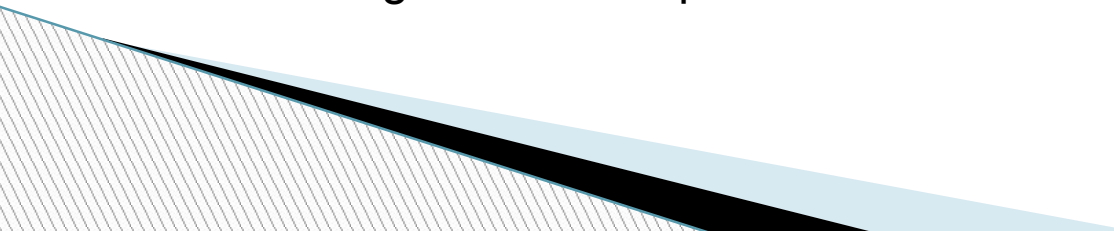
Collaborations

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

Consequences

Class and object adapters have different trade-offs.

A class adapter:

1. Adapts Adapter 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.
 2. Lets-adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
 3. Introduces only one object, and no additional pointer indirection is needed to get to the Adaptee
- 

Consequences

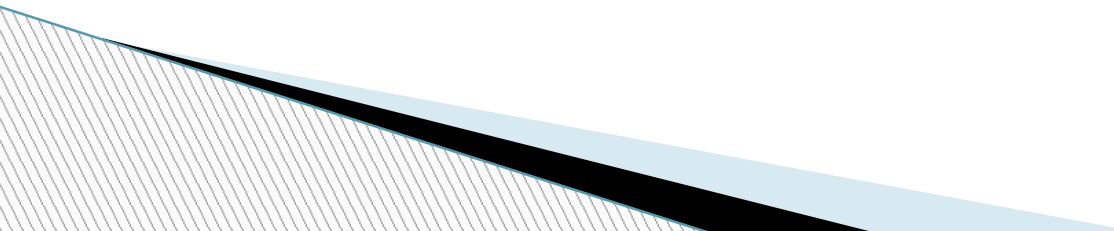
B Object adapter:

1. Lets 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.
2. Makes it harder to override Adaptee behavior. It will require sub-classing Adaptee and making Adapter refer to the subclass rather than the Adapter itself.

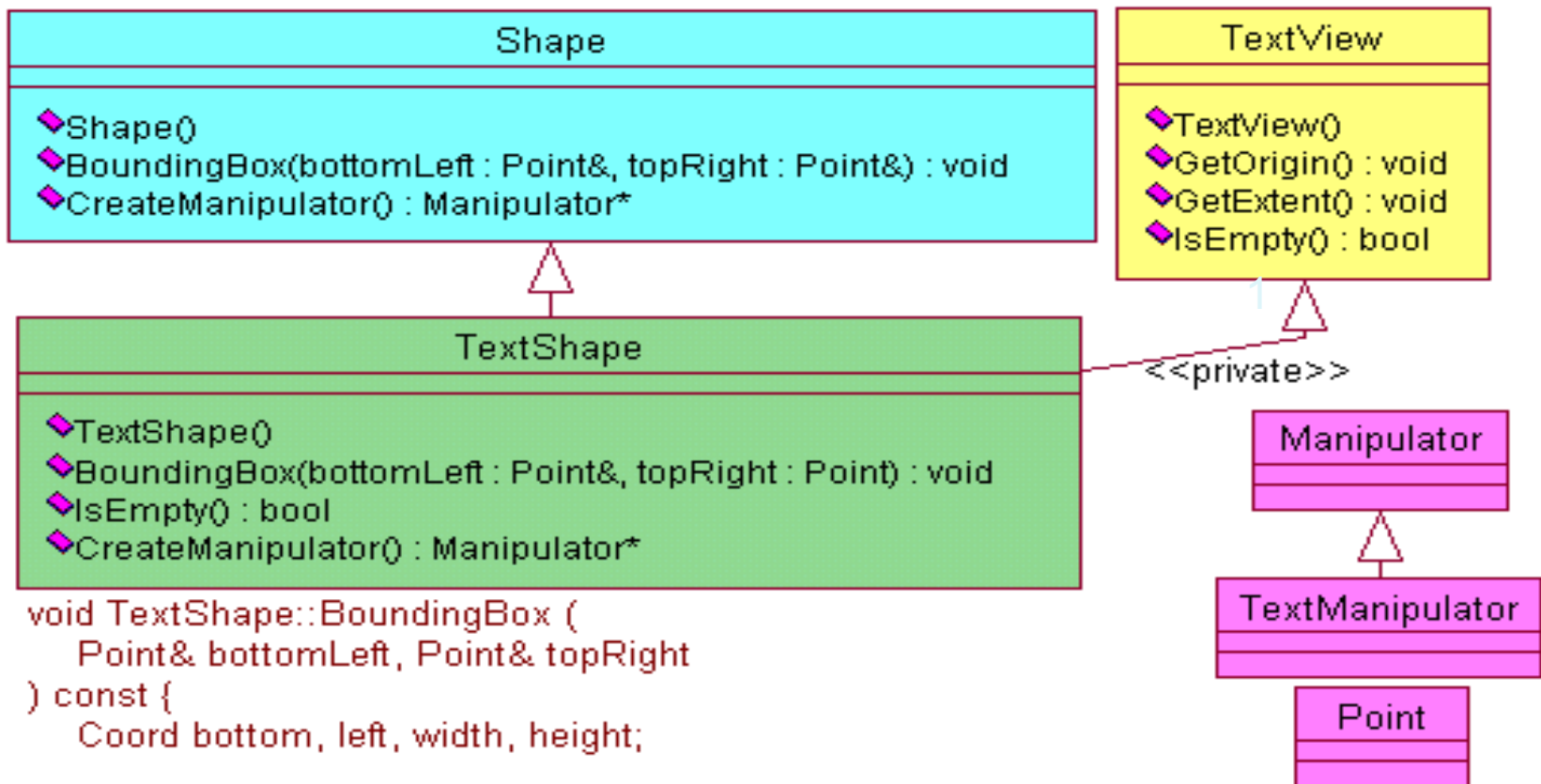
Issues to be considered while using adapter pattern:

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

Related pattern

- ▶ 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. A decorator is thus more transparent to the application than an adapter is. As a consequence, Decorator supports recursive composition, which isn't possible with pure adapter.
- 

Model: Adapter Class Pattern



```
void TextShape::BoundingBox (
    Point& bottomLeft, Point& topRight
) const {
    Coord bottom, left, width, height;

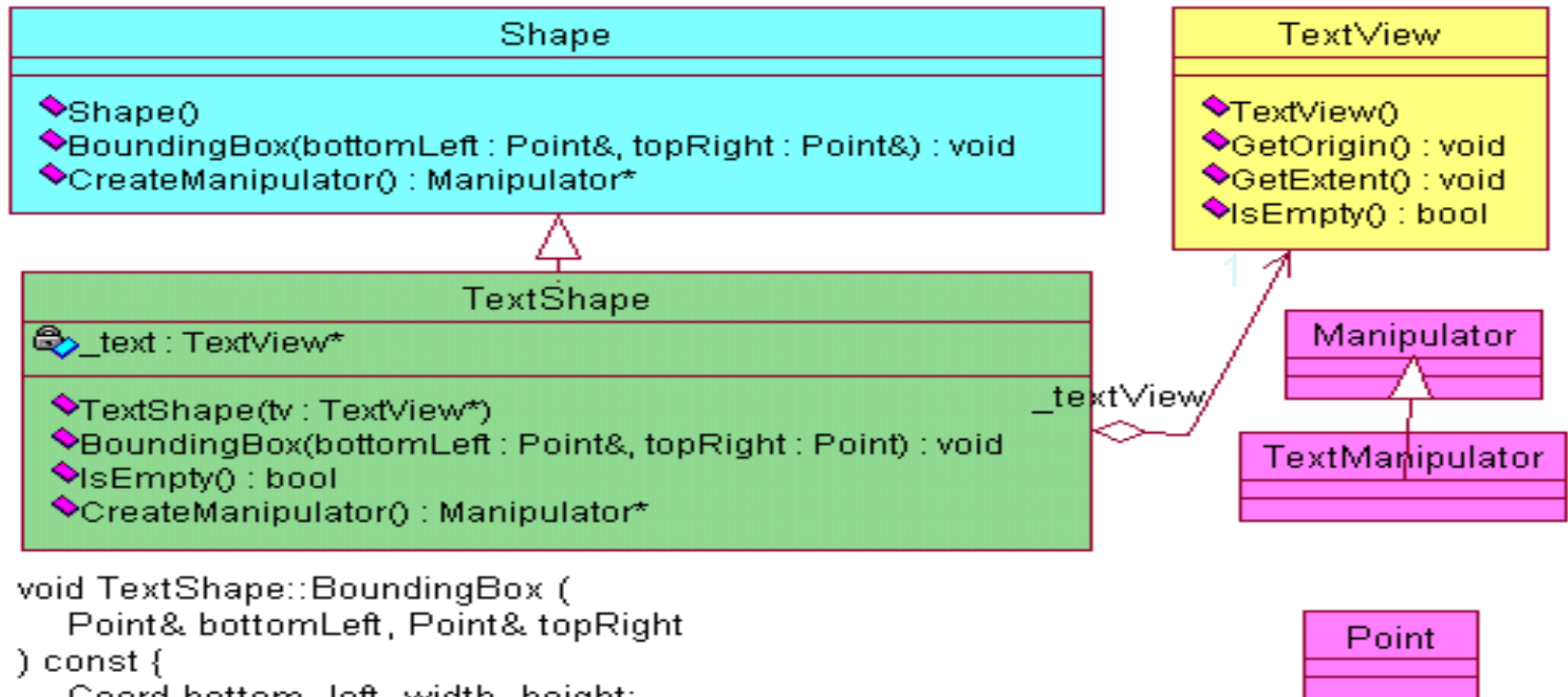
    GetOrigin(bottom, left); // from TextView
    GetExtent(width, height); // from TextView
```

```
    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
```

```
}
```

The TextShape::BoundingBox converts TextViews interface to conform to Shape's Interface

Model: Adapter *object* Pattern



```

void TextShape::BoundingBox (
    Point& bottomLeft, Point& topRight
) const {
    Coord bottom, left, width, height;

    _text->GetOrigin(bottom, left);
    _text->GetExtent(width, height);

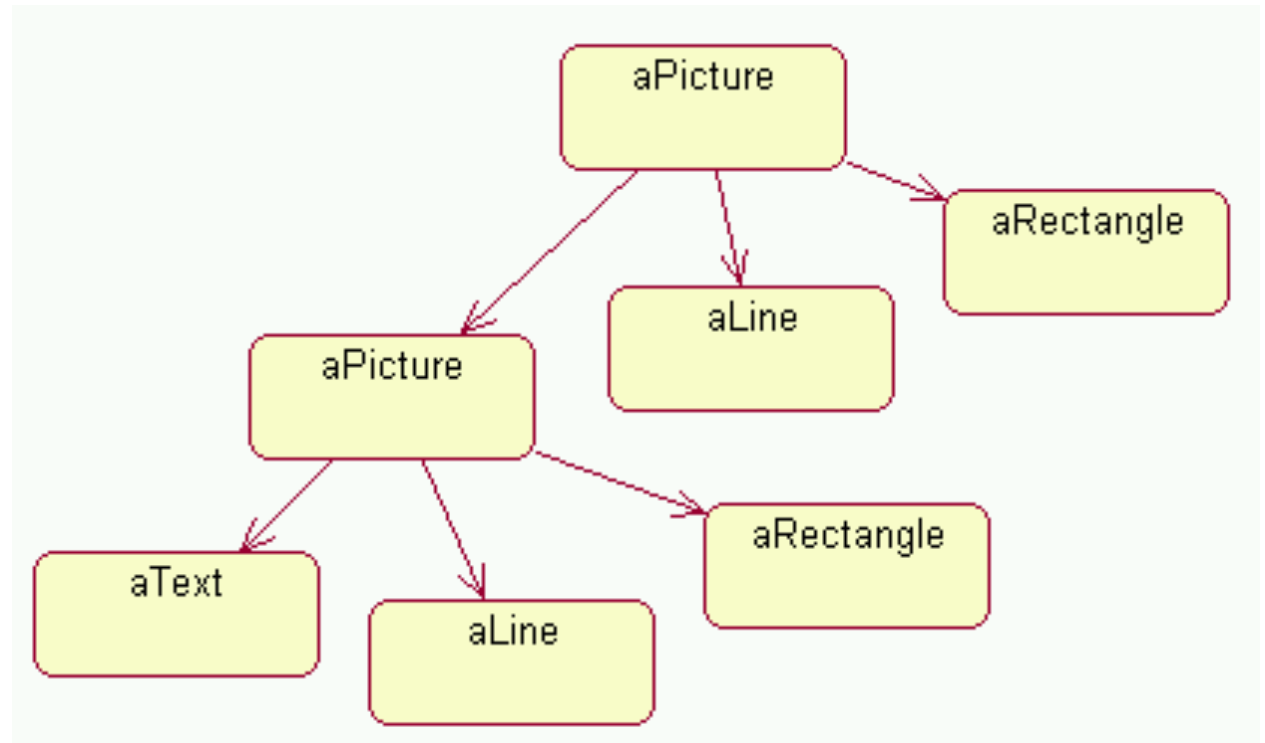
    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}
  
```

```

TextShape::TextShape (TextView* t) {
    _text = t;
}
  
```

Composite

Composite



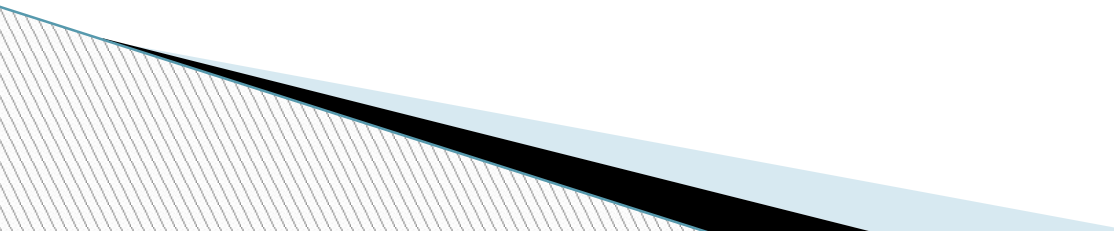
The *Composite* composes objects into tree structures, and lets clients treat individual objects and compositions uniformly.

Intent

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

Problem

Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to query the "type" of each object before attempting to process it is not desirable



Composite

- ▶ Example: figures in a structured graphics toolkit

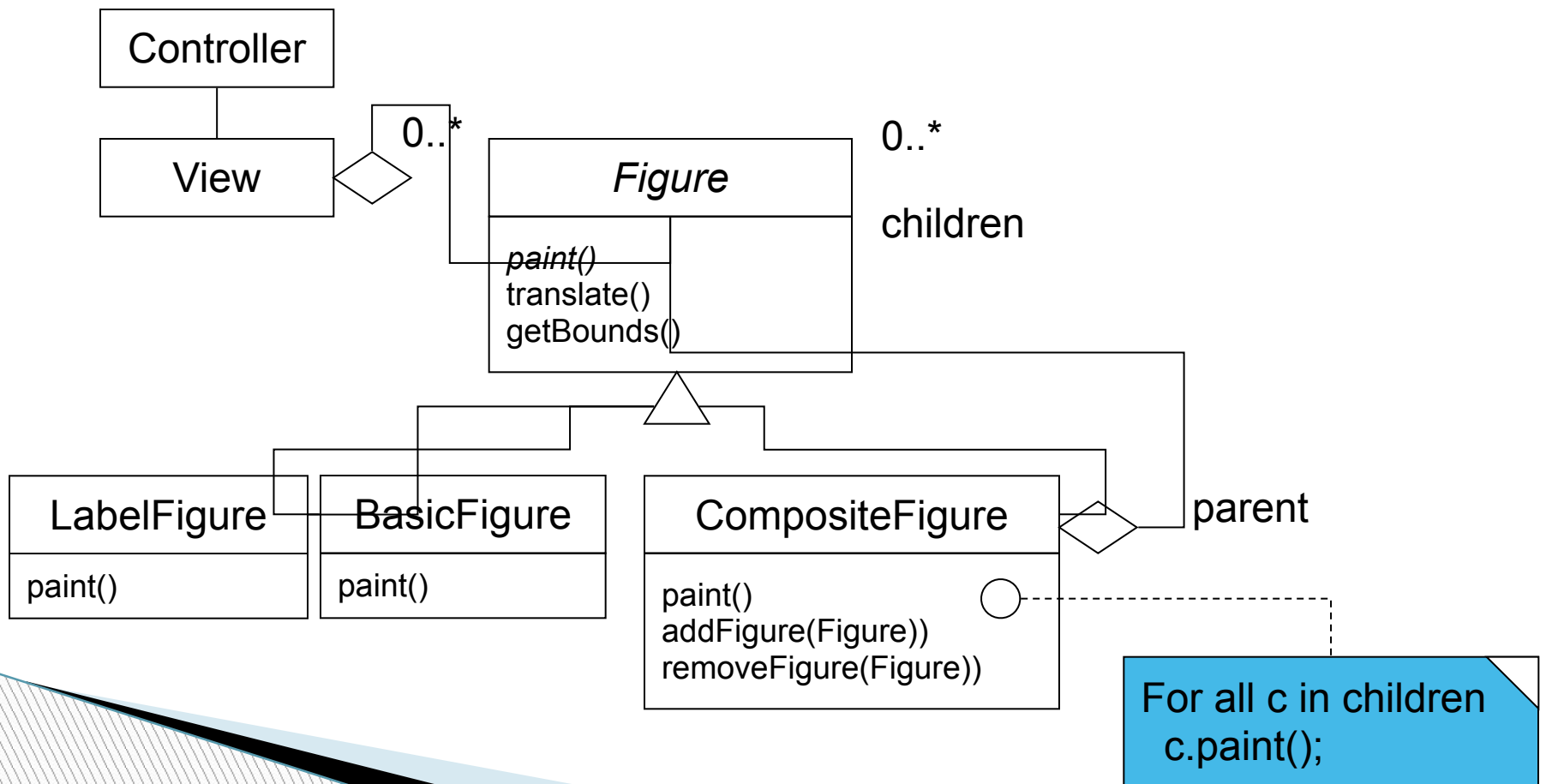
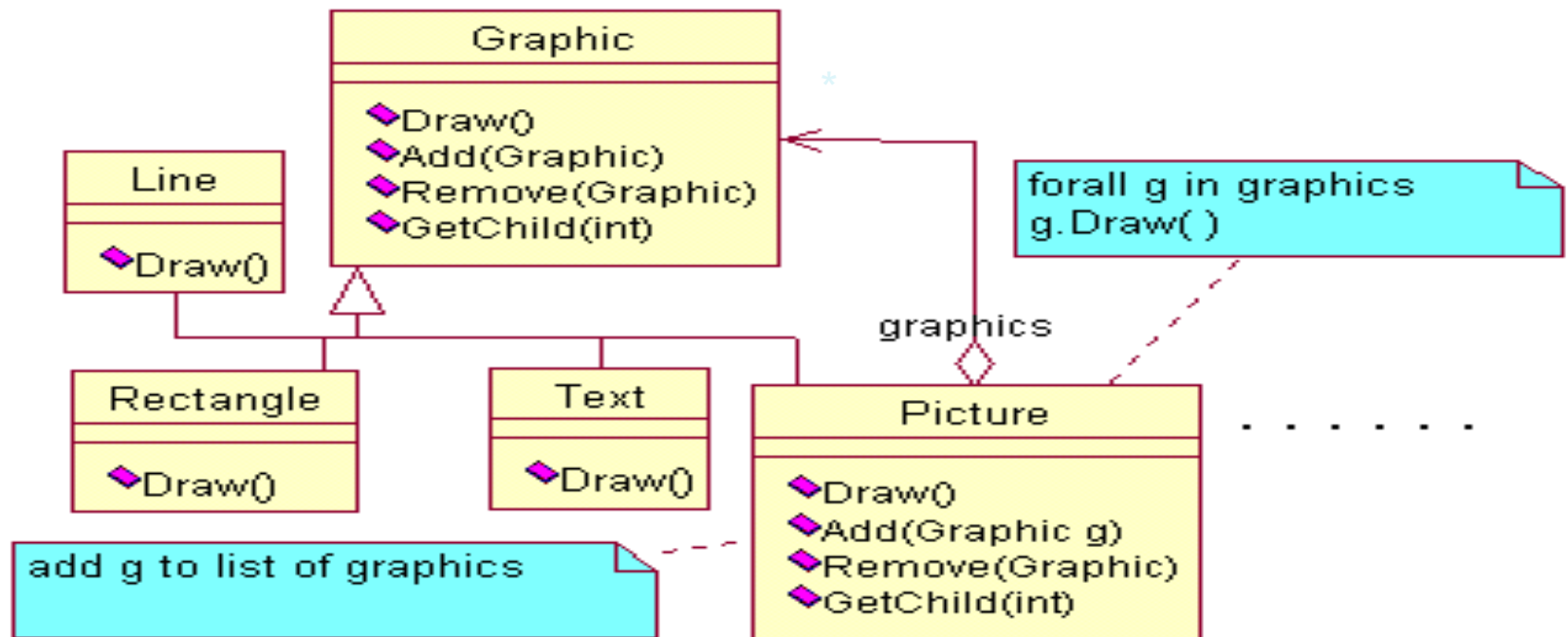


Illustration: Problem

- ▶ In situation where there is need of building a large complex components from the simple component as in Graphics applications Like Drawing editors and schematic capture system as given below in figure. Where the user can group components to form larger components which in



Illustration

Usually this can be implemented in simple way by defining the classes for graphical primitive such as Text and Lines plus other classes that act as containers for these primitive but this approach will cause problem:

“ Code that uses these classes must treat primitive and container objects differently, even though the user treats them identically”. To distinguish these objects will make the application more complex.

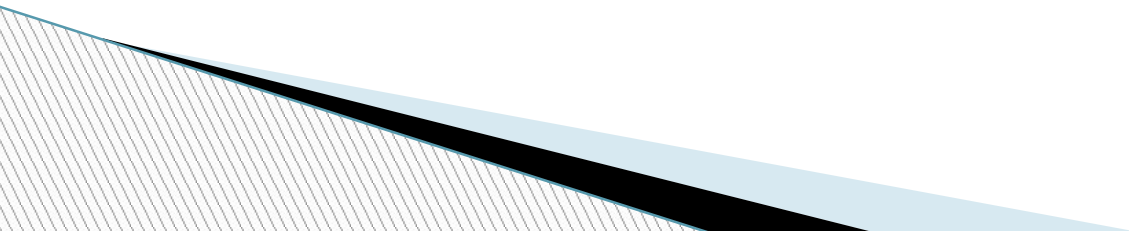
Solution:

The composite patterns describe how to use recursive composition so that the client's don't have to make this distinction. One Key aspect of the Composite patterns is an abstract class that represents both primitives and their containers.

Define an abstract base class (Component) that specifies the behavior that needs to be exercised uniformly across all primitive and composite objects.

Subclass the Primitive and Composite classes off of the Component class.

Each Composite object "couples" itself only to the abstract type Component as it manages its "children".

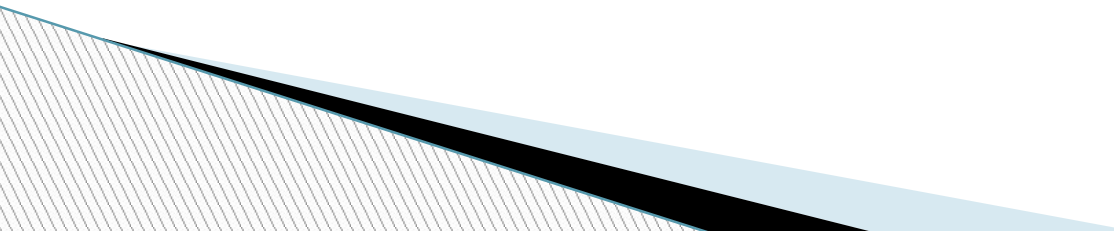


Applicability

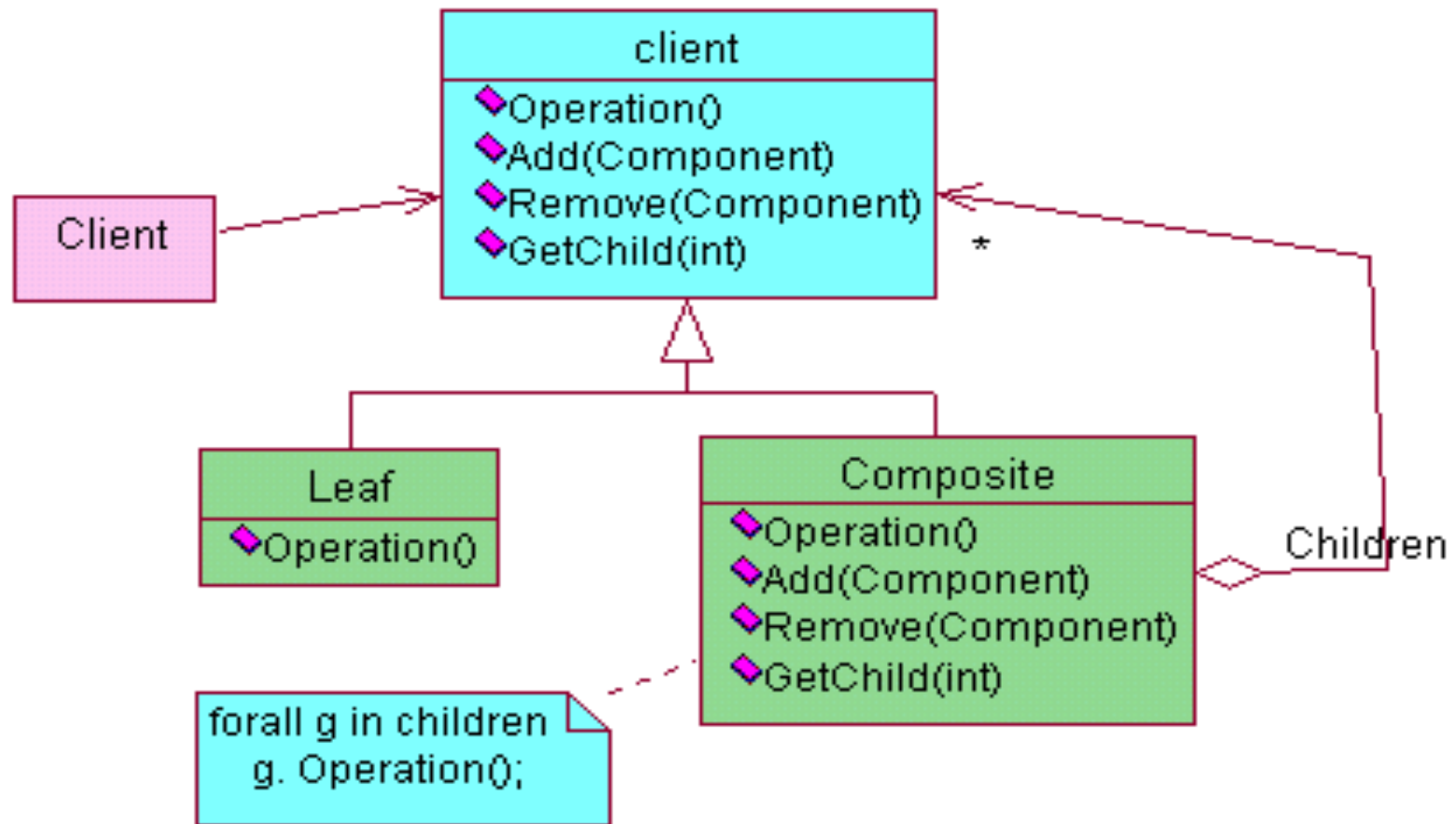
Use the Composite pattern when

You want to represent part-whole hierarchies of objects.

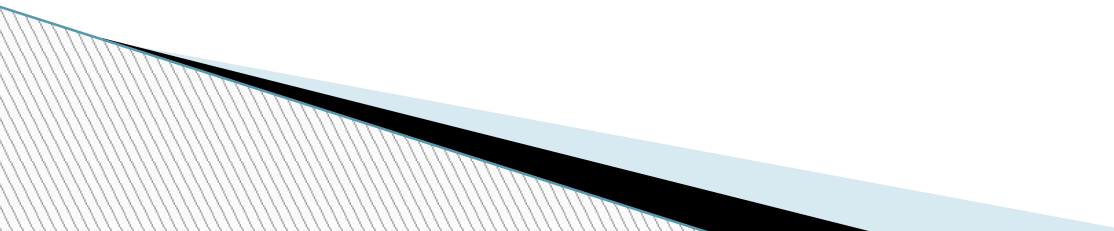
You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.



Structure



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

Defines class hierarchies consisting of primitive objects and composite objects recursively: Wherever client code expects a primitive object, it can also take a composite objects

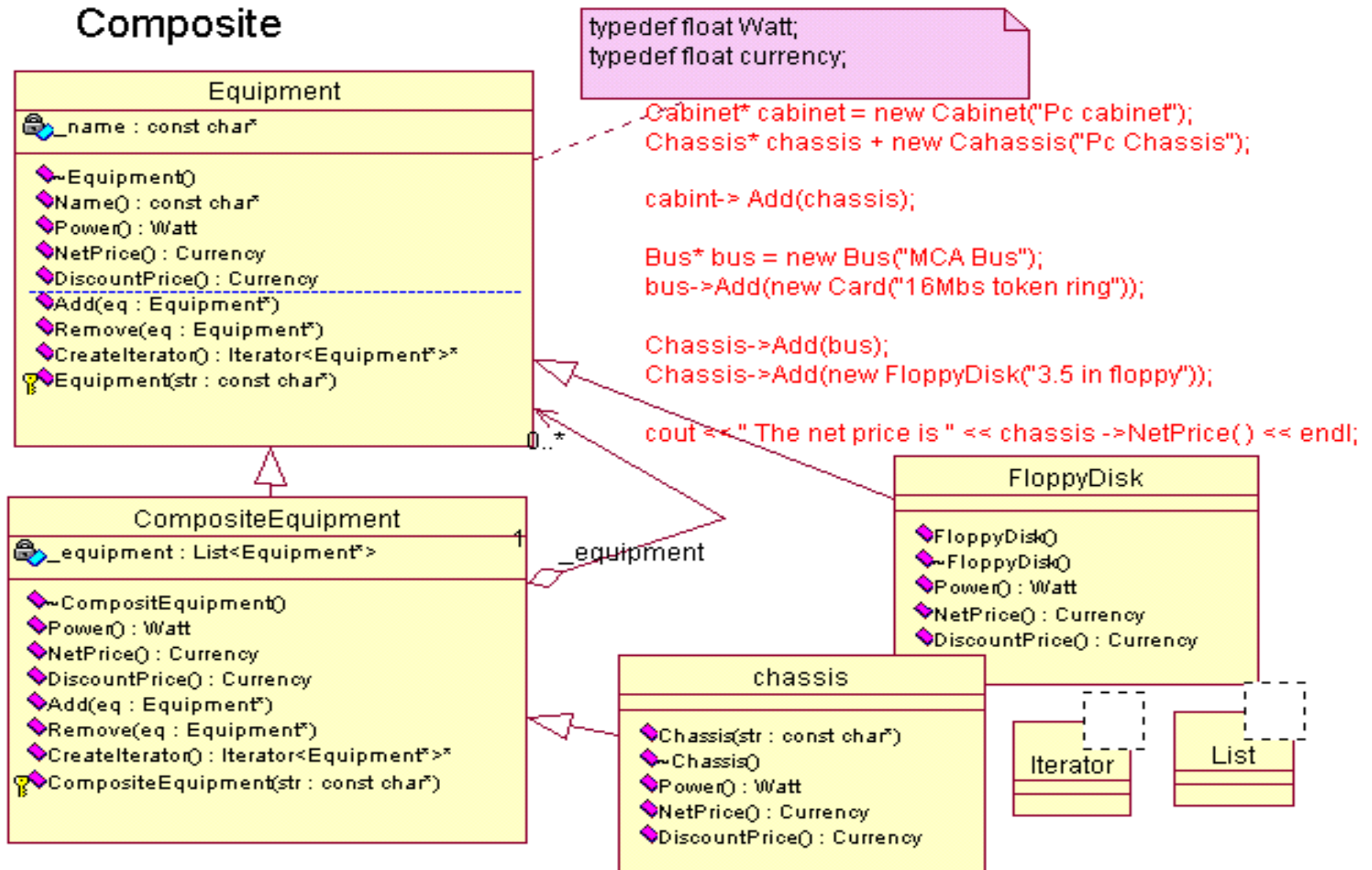
Make the client simple: clients treat composite and leaf uniformly. This simplifies client code, because it avoids having to write tag-and-case-statements-style functions over the classes that define the composition.

Makes it easier to add new kinds of components :Newly defined composition or leaf subclasses work automatically

Can make your design overly general : If you want composite to have only

Model

Composite



Related Patterns

Decorator 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.

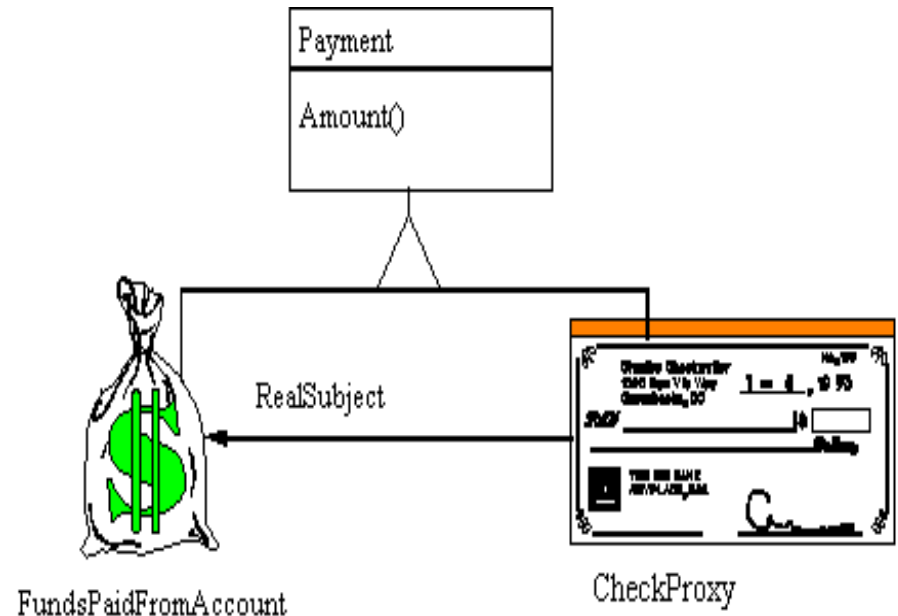
Iterator can be used to traverse composites.



Proxy

Proxy

Object Diagram for *Proxy* using Bank Draft Example



The ***Proxy*** provides a surrogate or place holder to provide access to an object. A check or bank draft is a proxy for funds in an account. A check can be used in place of cash for making purchases and ultimately controls access to cash in the issuer's account.

Proxy

Intent

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

Also Known As
Surrogate

Problem

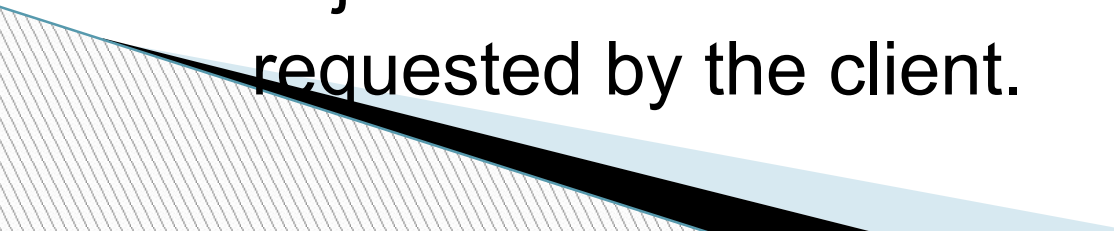
- You need to support resource-hungry objects, and you do not want to instantiate such objects unless and until they are actually requested by the client.
- 

Illustration :problem

Consider a document editor (Html creator) that can embed graphical objects in a document. Some graphical objects, like large raster images, can be expensive to create. But opening a document should be fast, so we should avoid creating all the expensive objects at once when the document is opened. This isn't necessary anyway, because not all of these objects will be visible in the document at the same time.

These constraints would suggest creating each expensive object **on demand**, which in this case occurs when an image becomes visible.

But what do we put in the document in place of the image? And how can we hide the fact that the image is created on demand so that we don't complicate the editor's implementation?

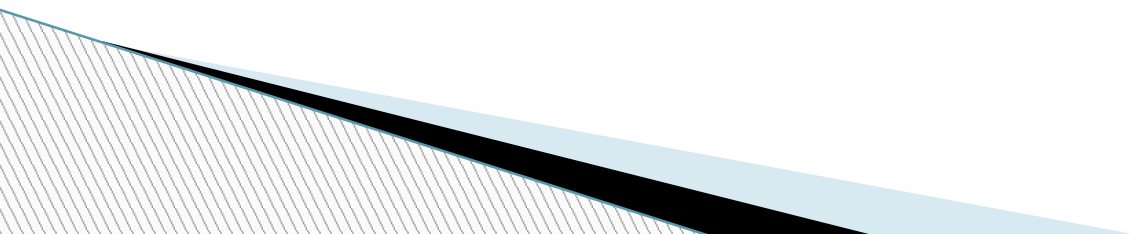
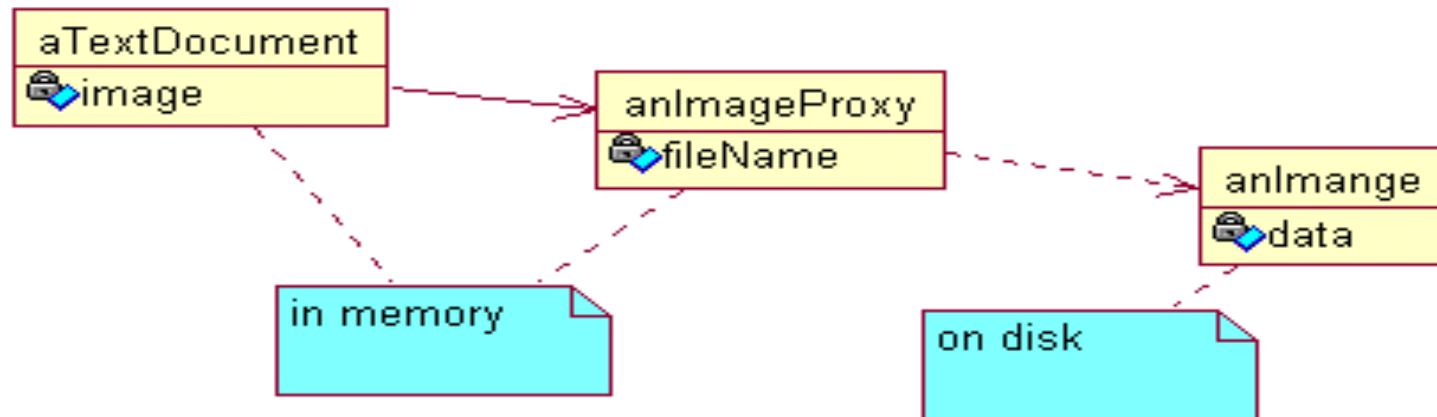


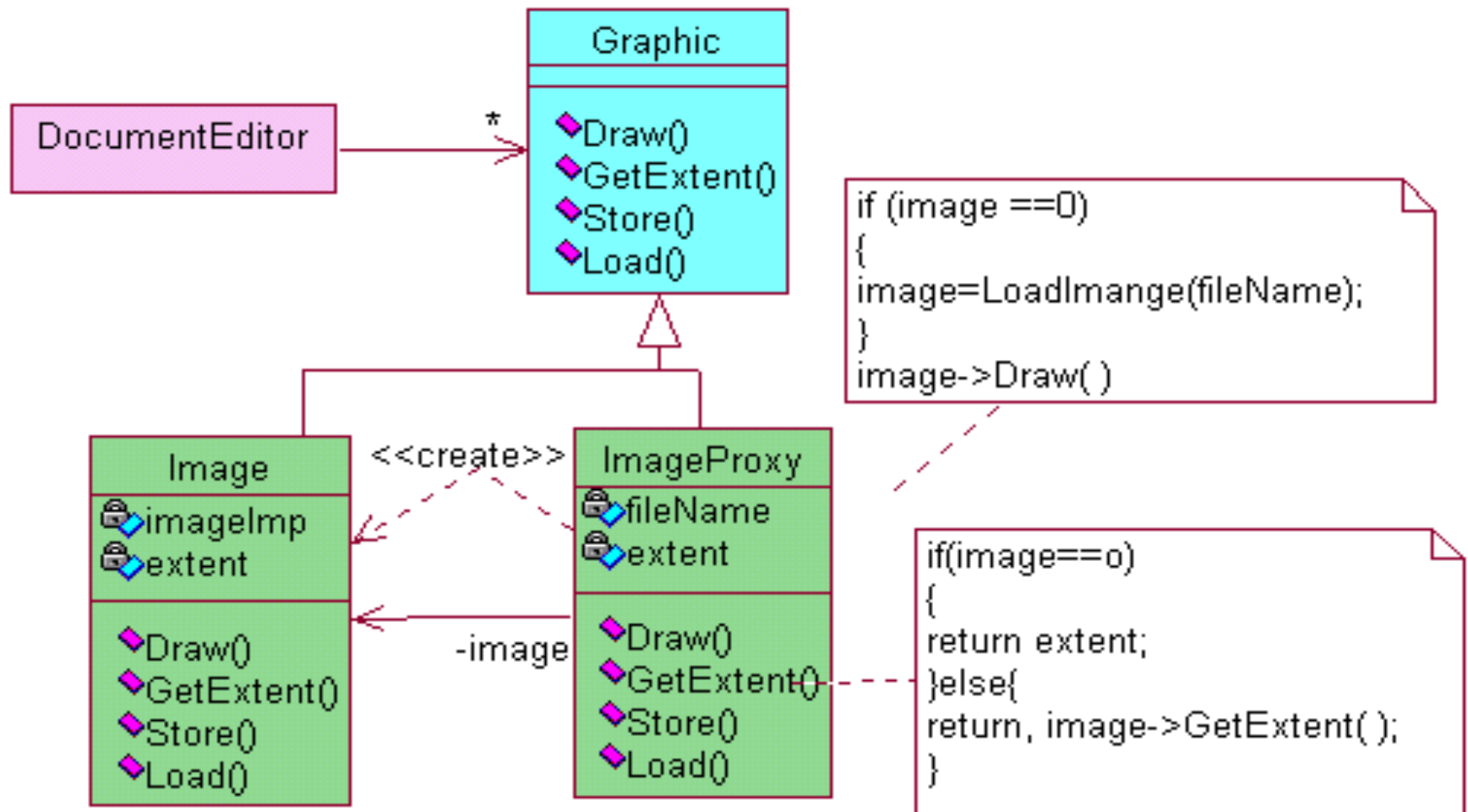
Illustration: solution

- The solution is to use another object, an image proxy, stand-in for the real image.
 - The **proxy** acts just like the image and takes care of when it's required.
- that acts as a
instantiating it



- 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.

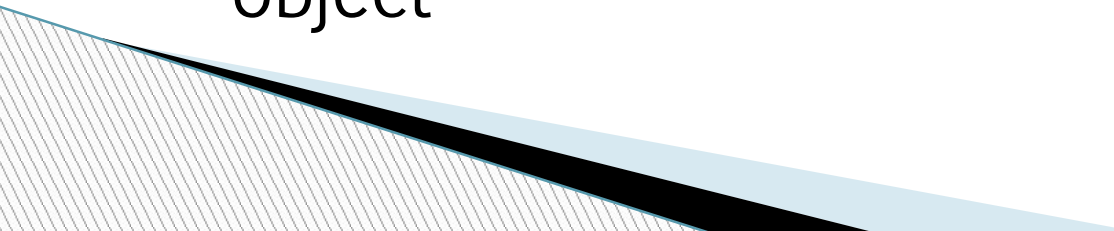
Illustration: example Document Editor



Solution

Design a surrogate, or proxy, object that: instantiates the real object the first time the client makes a request of the proxy, remembers the identity of this real object, and forwards the instigating request to this real object.

Then all subsequent requests are simply forwarded directly to the encapsulated real object



Applicability

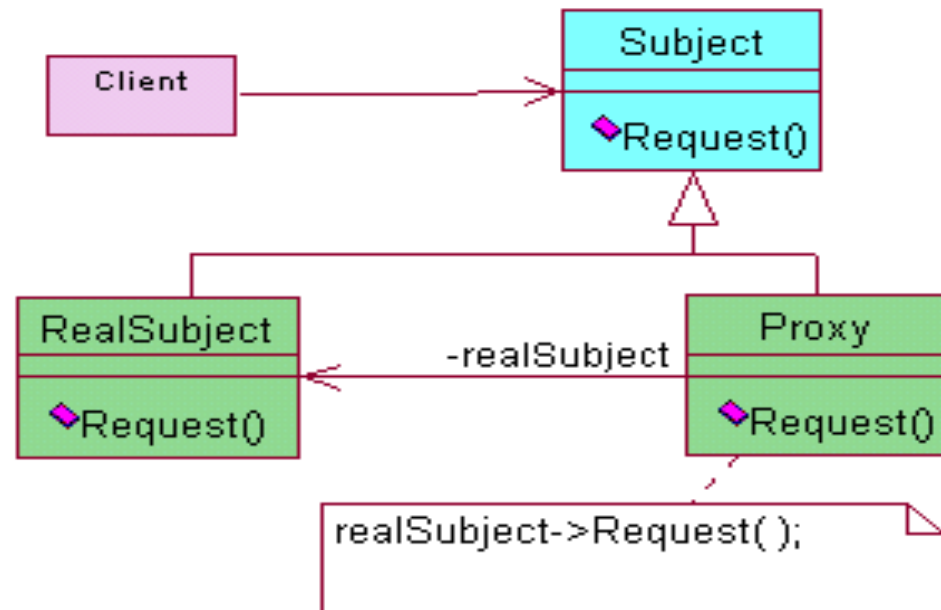
Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. Here are several common situations in which the Proxy pattern is applicable:

1. A **remote proxy** provides a local representative for an object in a different address space. NEXTSTEP uses the class NXProxy for this purpose.
2. A **virtual proxy** creates expensive objects on demand. The Image-Proxy described in the Motivation is an example of such a proxy.

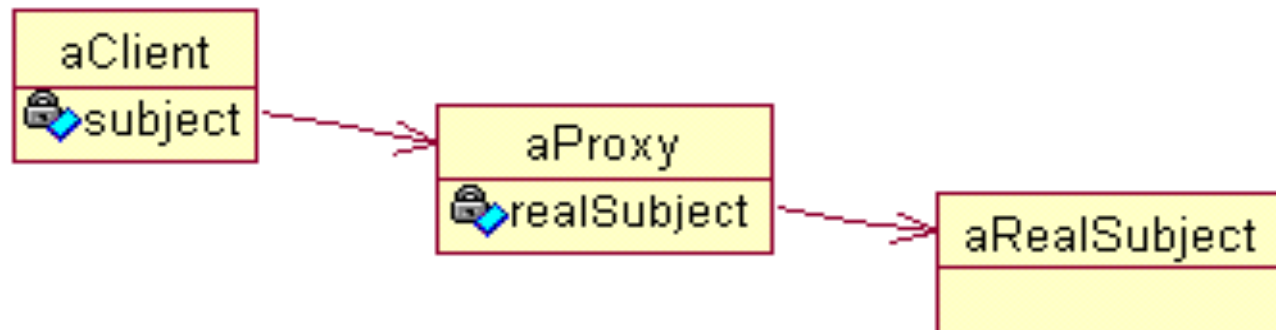
Applicability

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

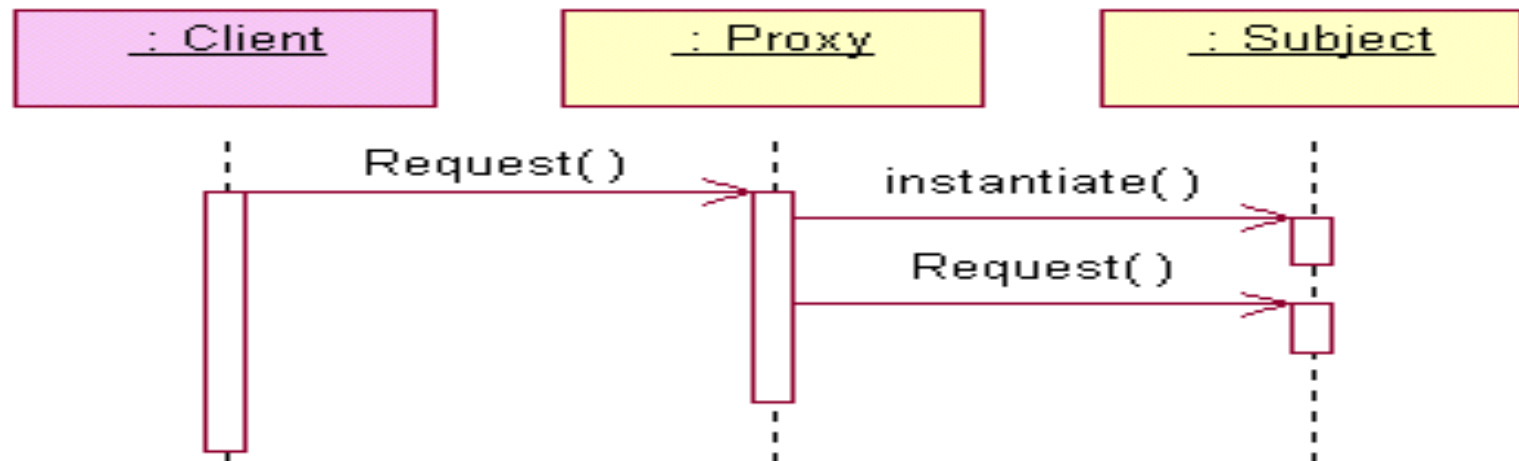


Object diagram of a proxy structure at run-time:



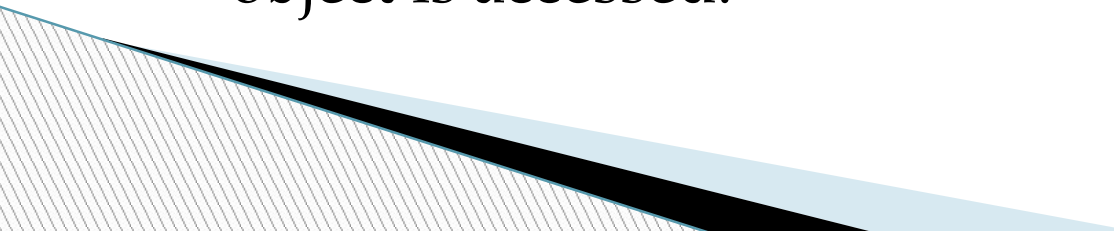
Collaborations

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

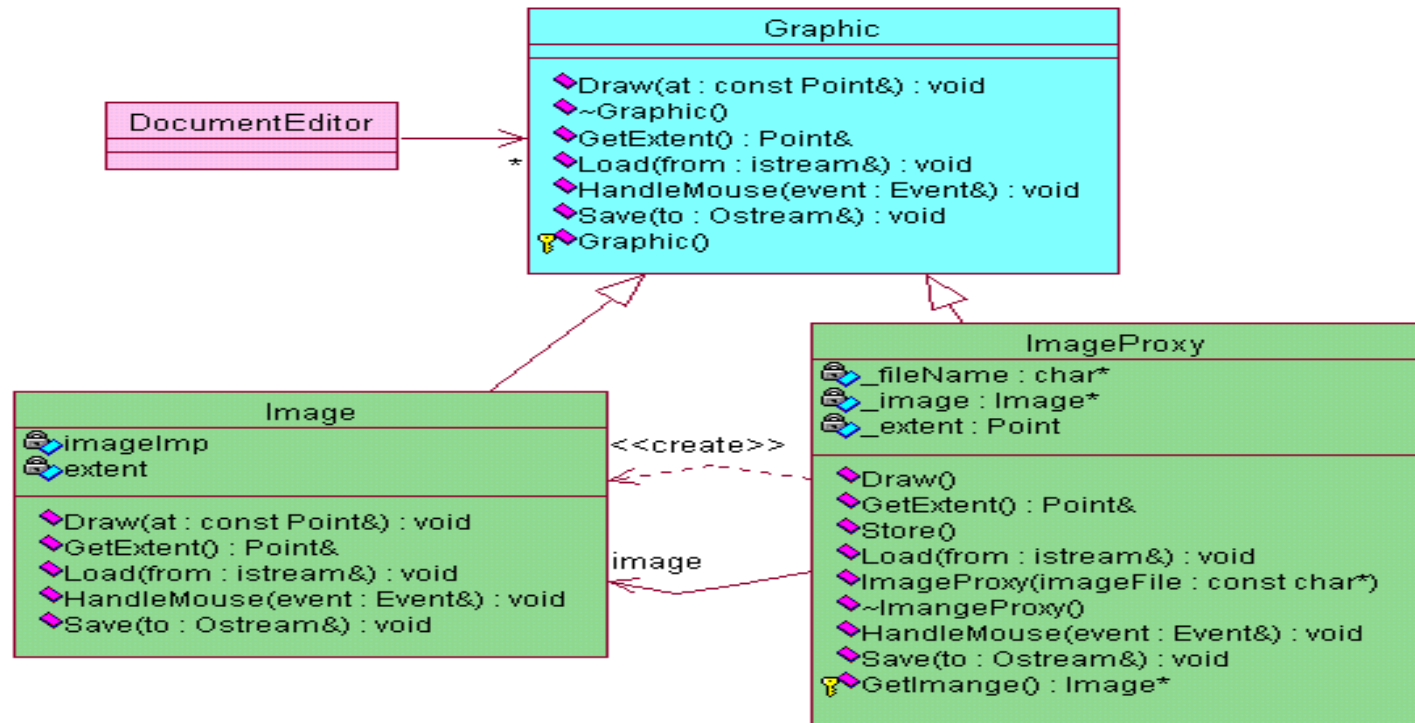


Consequences

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.
- 

Model



```

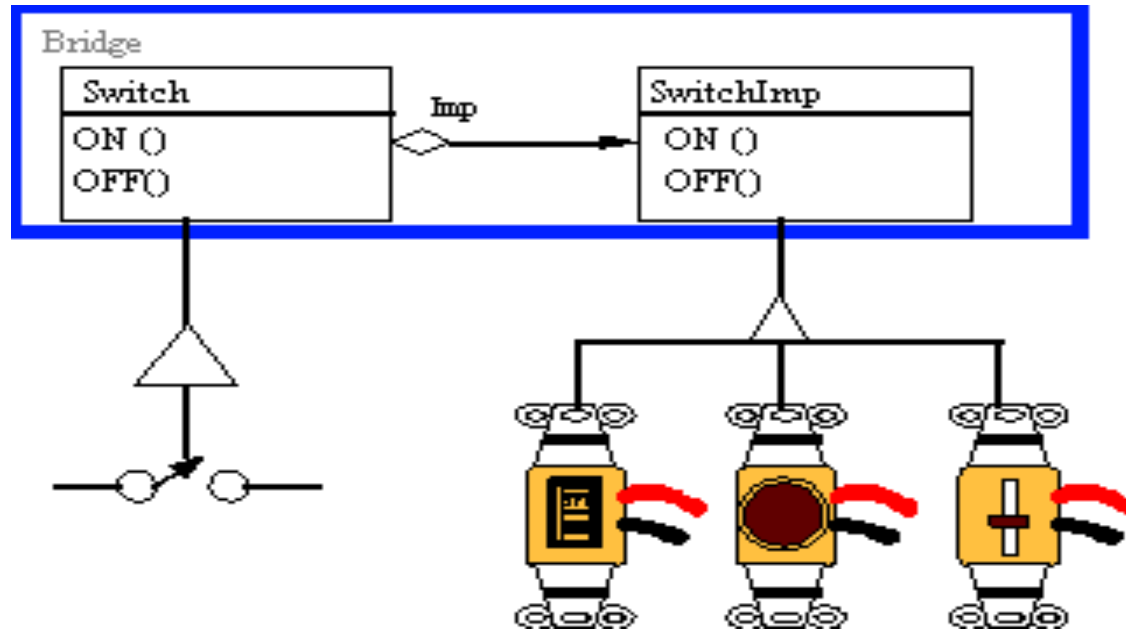
ImageProxy::ImageProxy (const char* fileName) {
    _fileName = strdup(fileName);
    _extent = Point::Zero; // don't know extent yet
    _image = 0;
}

Image* ImageProxy::GetImage() {
    if (_image == 0) {
        _image = new Image(_fileName);
    }
    return _image;
}
    
```

Bridge

Bridge

Object Diagram for *Bridge*
using Electrical Switch
Example



The *Bridge* pattern decouples an abstraction from its implementation, so that the two can vary independently. A household switch controlling lights, ceiling fans, etc. is an example of the *Bridge*. The purpose of the switch is to turn a device on or off. The actual switch can be implemented as a pull chain, a simple two position switch, or a variety of dimmer switches

Intent

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

Also Known As
Handle/Body

Problem

"Hardening of the software arteries" has occurred by using subclassing of an abstract base class to provide alternative implementations. This locks in compile-time binding between interface and implementation. The abstraction and implementation cannot be independently extended or composed.

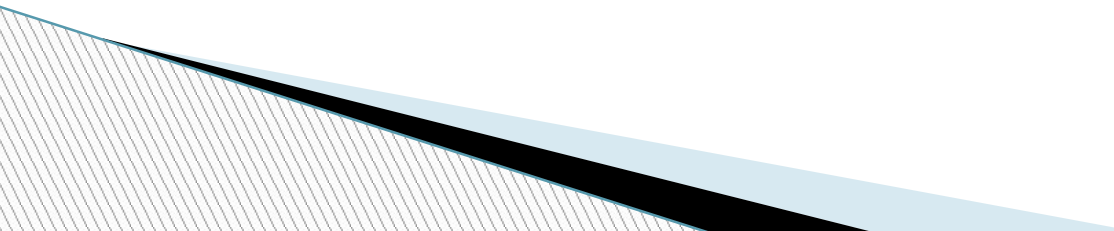


Illustration: Problem

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 x window system and IBM's PM .

Using inheritance, its possible to define abstract class Window and subclasses Xwindows and PM that implement the Window interface for the different platforms. But this has 2 drawbacks.

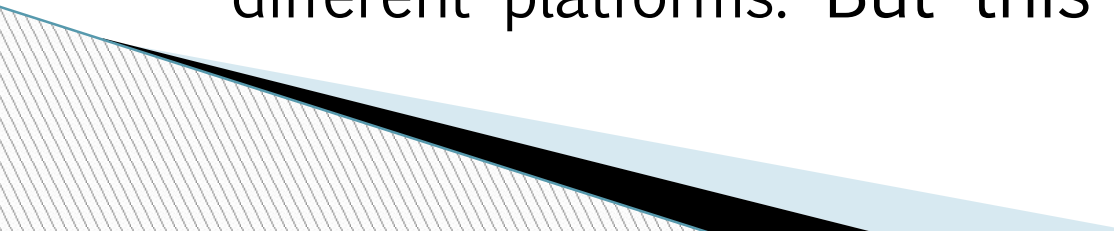


Illustration: Problem

Draw Backs:

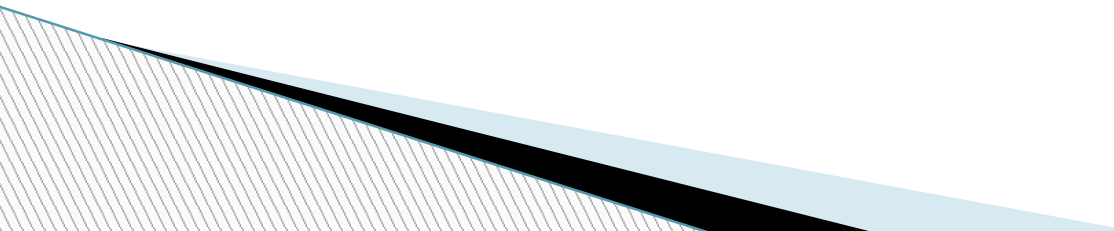
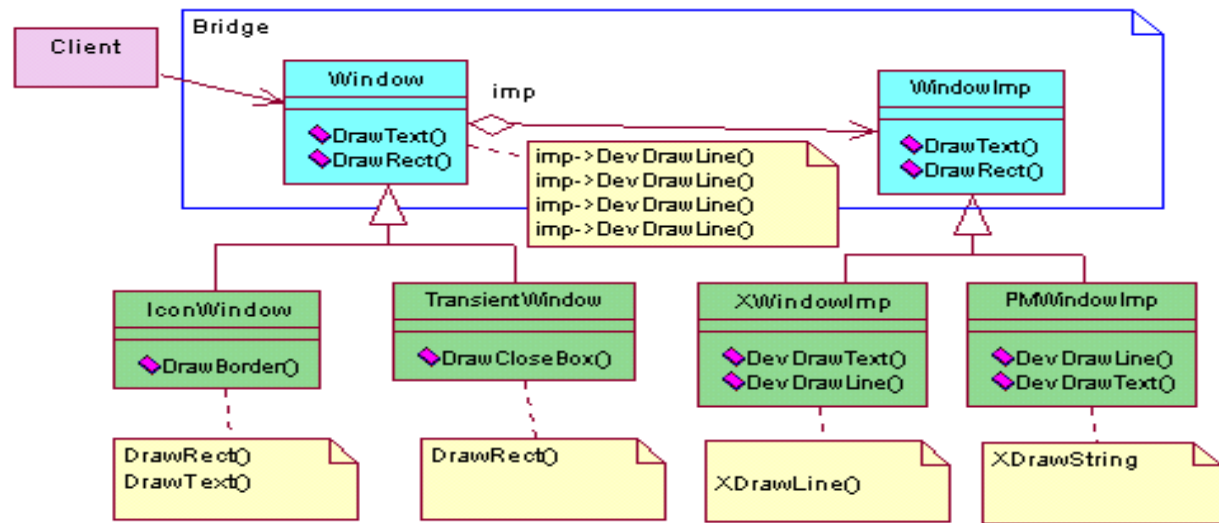
- 1> It's inconvenient to extend the Window abstraction to cover different kinds of windows or new platform (ie. Every new platform a new Window subclass for every Kind of Window is necessary)
 - 2> It makes client code platform-dependent. Whenever a client creates a window, it instantiates a concrete class that has specific implementation. This, in turn, makes it harder to port the client code to other platforms.
- 

Illustration: Solution



Bridge patterns address 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 separate hierarchy for platform-specific window implementations, with **WindowImp** as its root.

All operation 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.

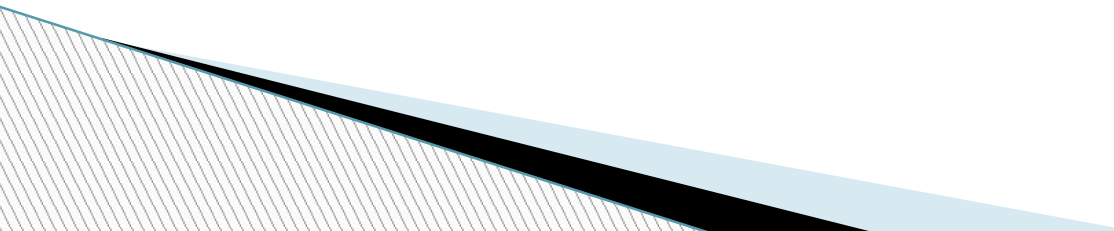
Solution:

Decompose the component's interface and implementation into orthogonal class hierarchies.

The interface class contains a pointer to the abstract implementation class. This pointer is initialized with an instance of a concrete implementation class, but all subsequent interaction from the interface class to the implementation class is limited to the abstraction maintained in the implementation base class.

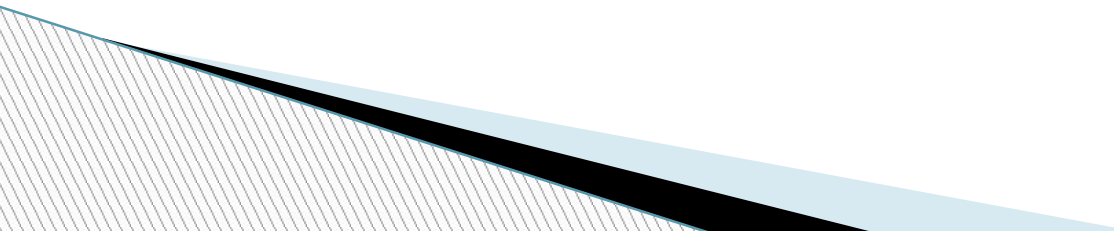
The client interacts with the interface class, and it in turn "delegates" all requests to the implementation class.

The interface object is the "handle" known and used by the client; while the implementation object, or "body", is safely encapsulated to ensure that it may continue to evolve, or be entirely replaced (or shared at run-time

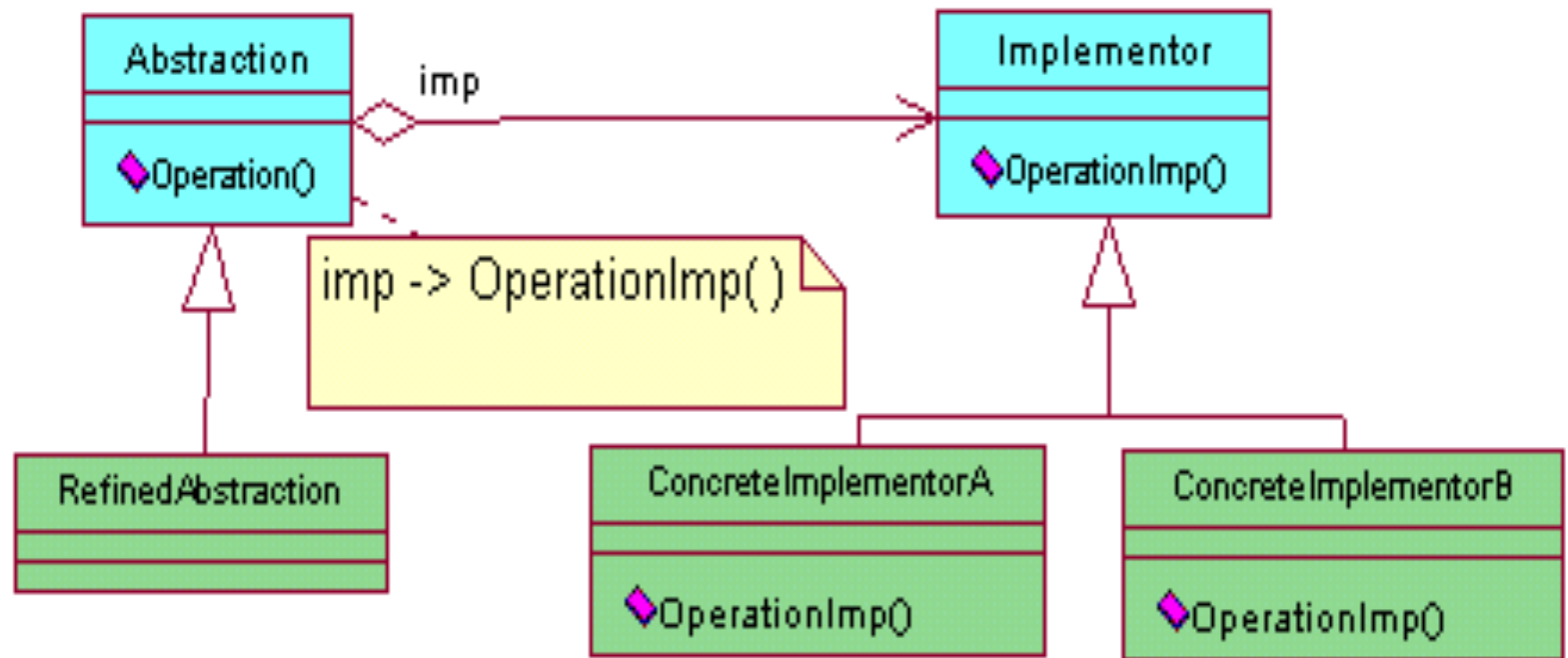


Applicability

► **Use the Bridge pattern when:**

- you want run-time binding of the implementation and want to avoid permanent binding between an abstract and its implementation.
 - Both the abstraction and their implementations should be extensible by subclassing.
 - Change in the implementation of an abstraction should have no impact on client; ie their code should not be recompiled
 - you want to share an implementation among multiple objects,
 - you have a proliferation of classes resulting from a coupled interface and numerous implementations
 - you need to map orthogonal class hierarchies.
- 

Structure



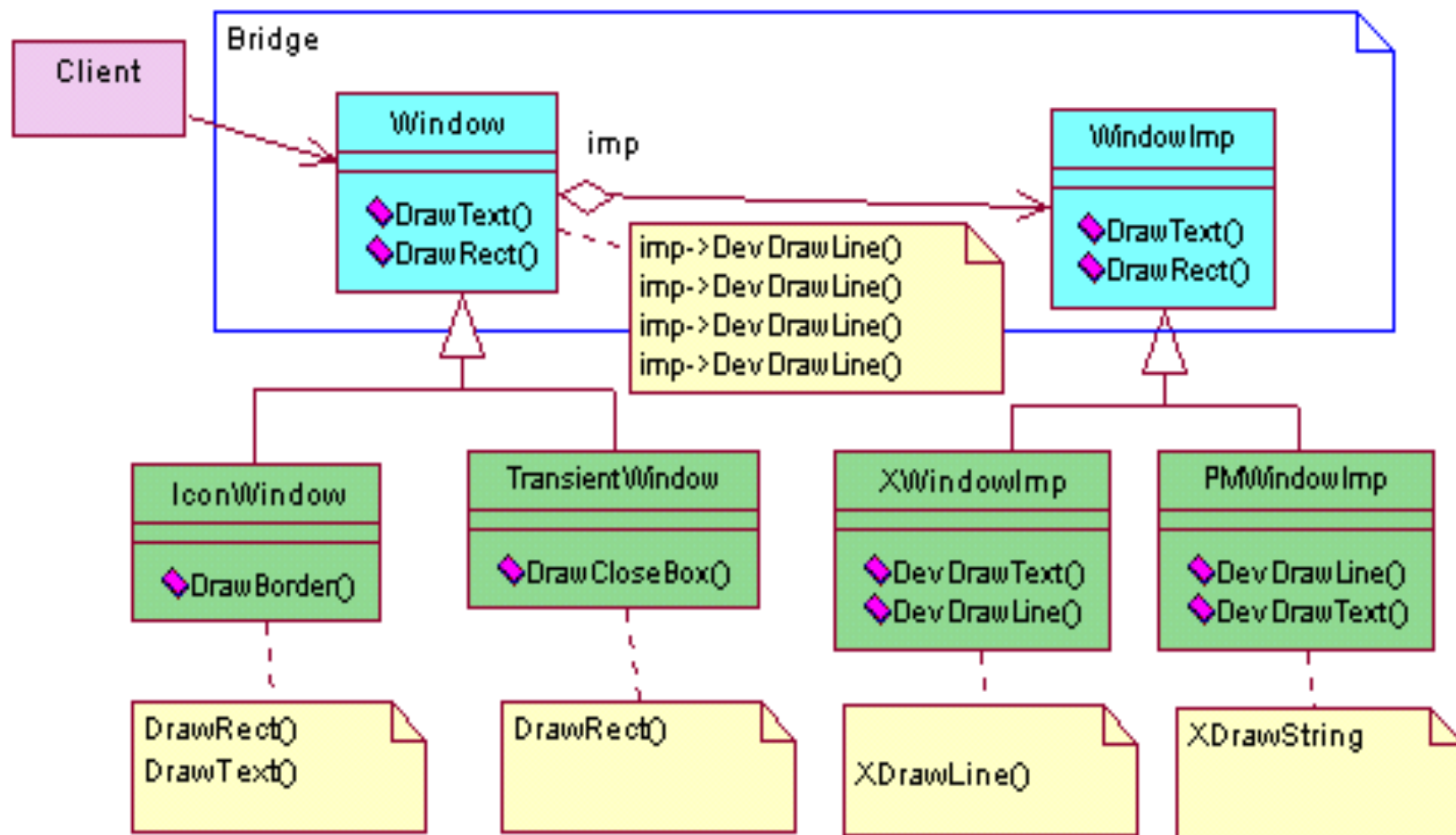
Collaborations

Abstraction forwards client request to its implementers object.

Consequences include:

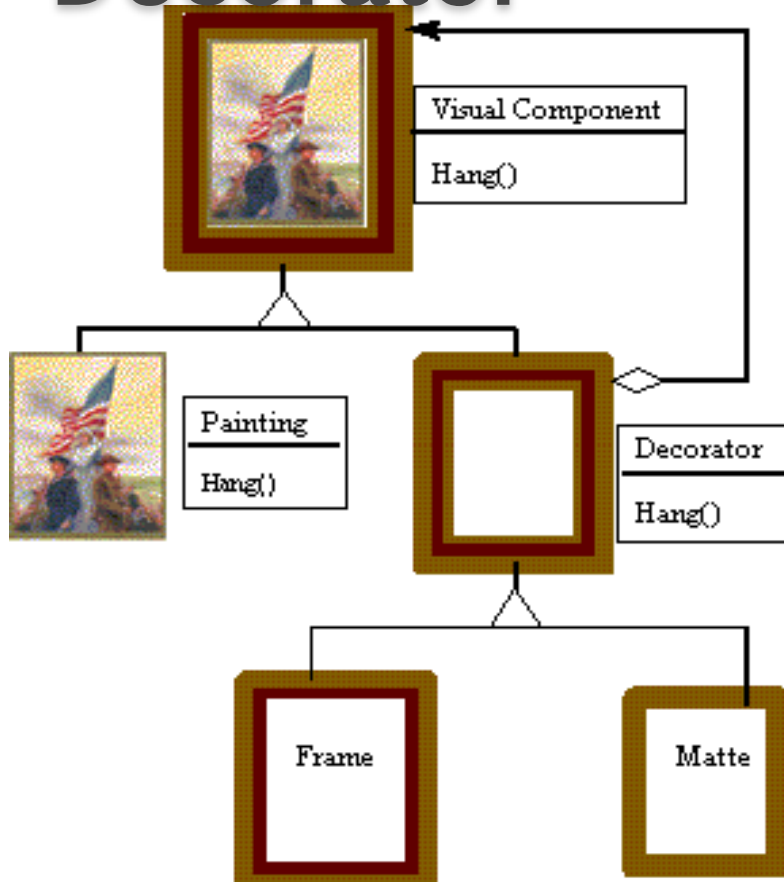
- decoupling the object's interface & implementation
- improved extensibility (you can extend (i.e. subclass) the abstraction and implementation hierarchies independently),
- hiding details from clients. Clients are hidden from the implementation details, like sharing of implementers objects and the accompanying reference count(in c++) mechanism(if any)

Model:



Decorator

Decorator



The *Decorator* attaches additional responsibilities to an object dynamically. Although paintings can be hung on a wall with or without frames, frames are often added, and it is the frame which is actually hung on the wall. Prior to hanging, the paintings may be matted and framed, with the painting, matting, and frame forming a single visual component.

Object Diagram for *Decorator* using Framed Painting Example

Intent

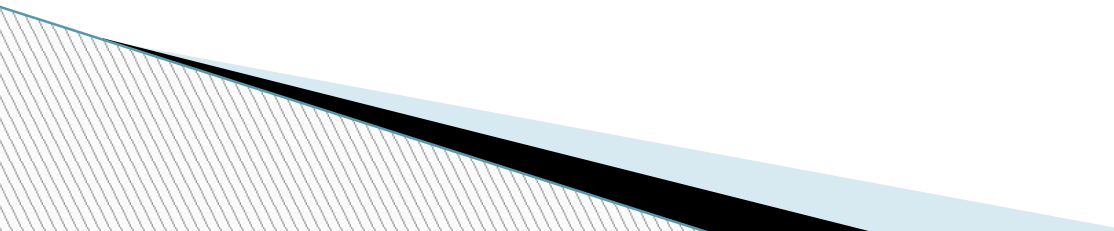
Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Alternative Name:

Wrapper

Problem

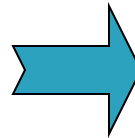
Sometimes we want to add responsibilities to individual objects (aTextView) not to an entire class. (i.e. how to give control to client to decorate the component with a border / a scroll bar)



Illustration

aTextView

Attached additional responsibility to an object dynamically. Decorator provide a flexible alternative to sub classing for extending functionality



Attached additional responsibility to an object dynamically. Decorator provide a flexible alternative to sub classing for extending functionality

aScrollDecorator

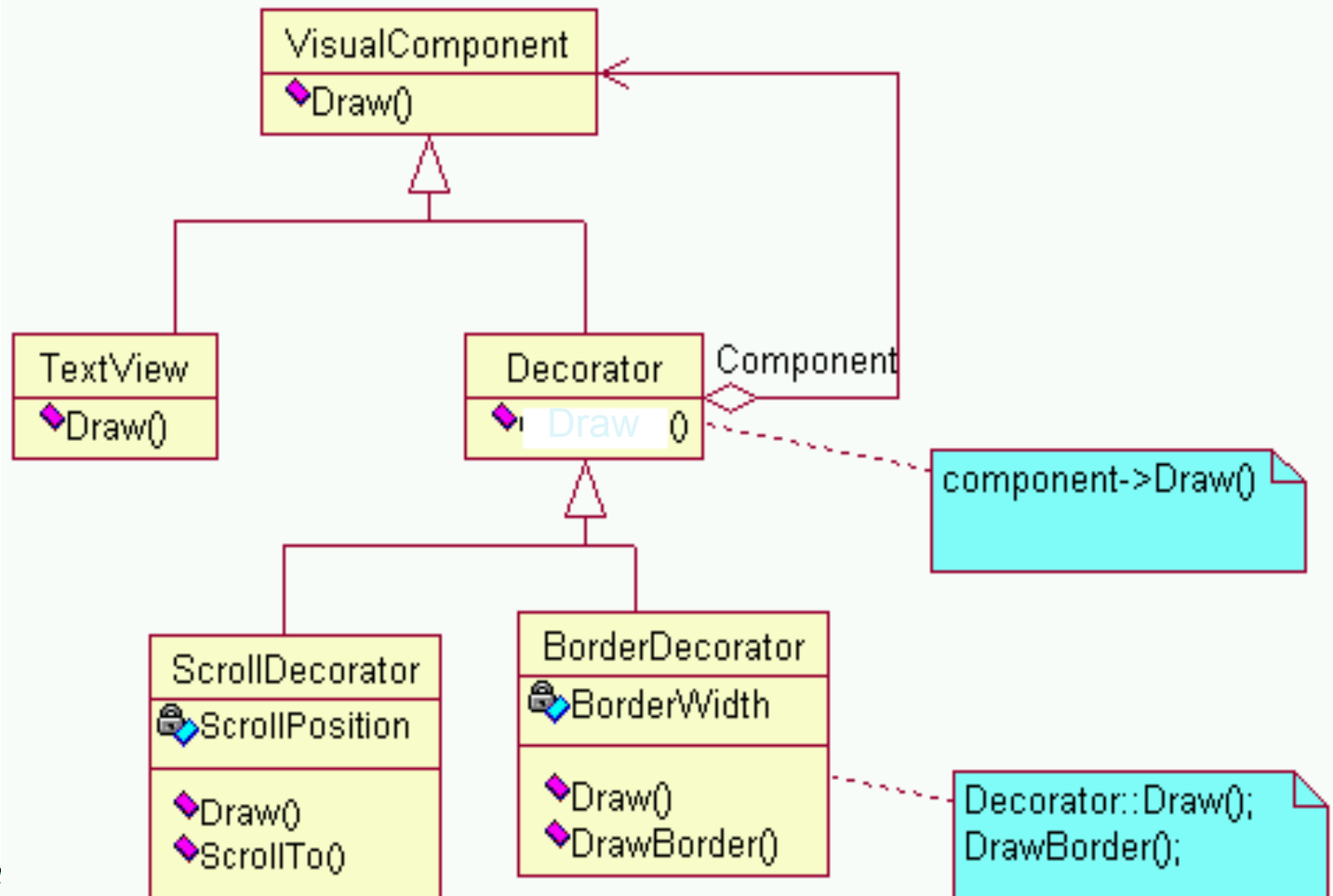
aBorderDecorator
component

aScrollDecorator
component

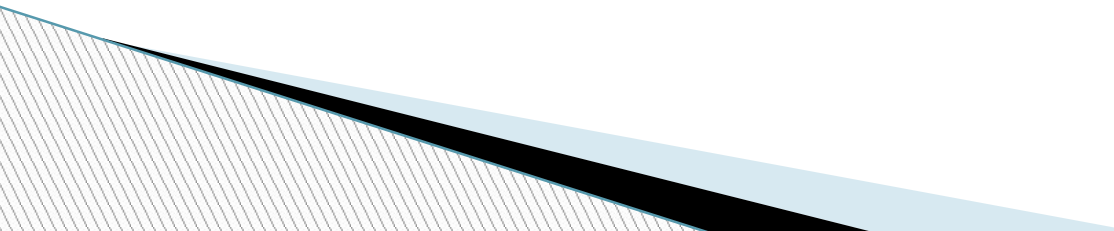
aTextView



Illustration

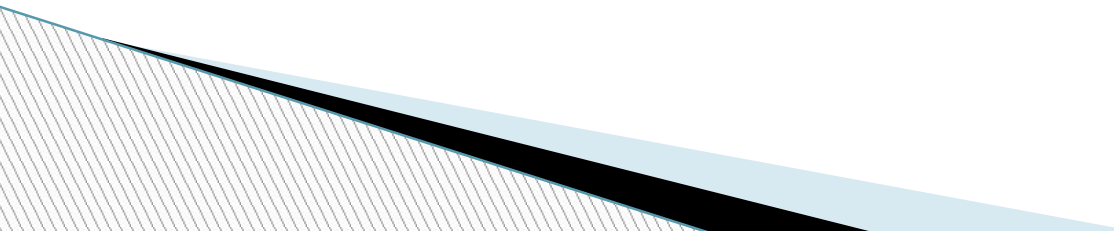


Solution

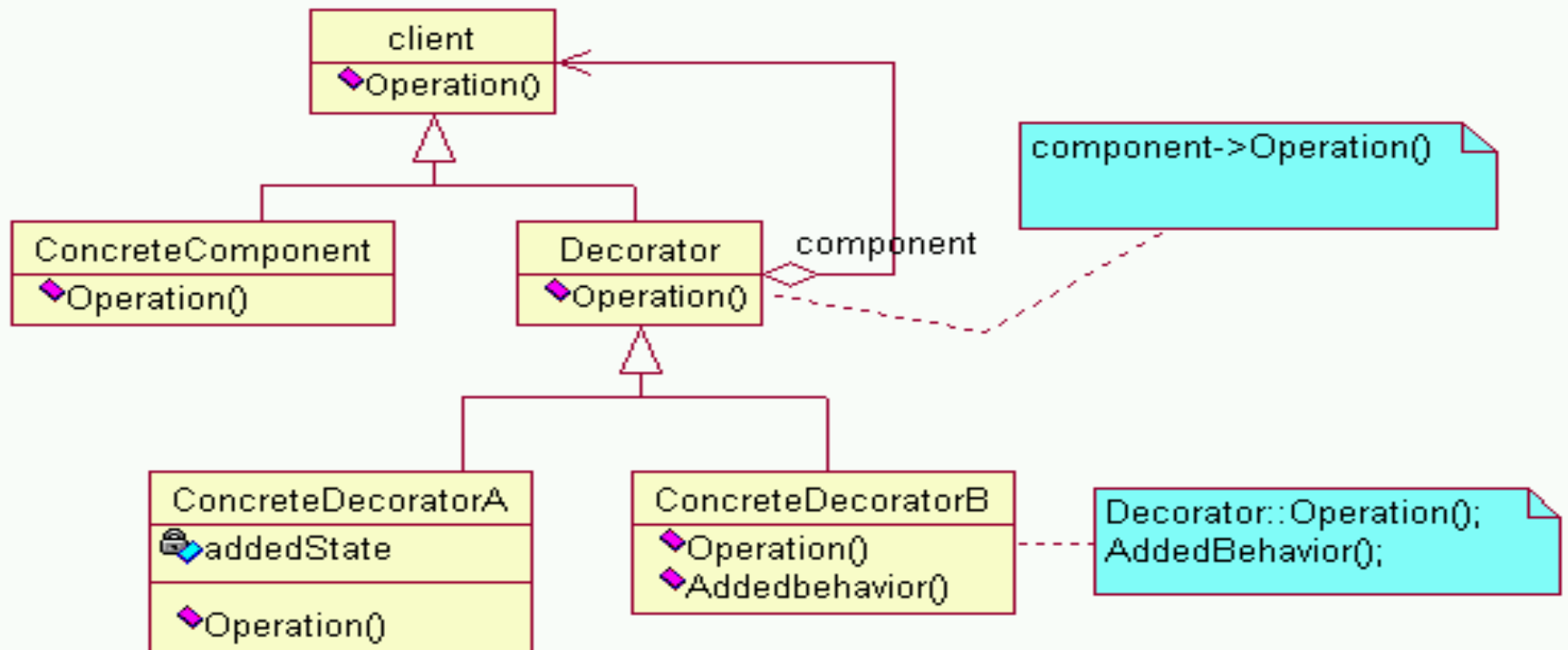
- ▶ One way to add responsibilities is with inheritance. Inheriting a border with another class puts a border along every subclass instance (inflexible solution).
 - ▶ A more flexible approach is to enclose the component in another object that adds the border. The enclosing object is called *Decorator*.
 - ▶ The *Decorator* conforms to interface of the component it decorates so that its presence is transparent to its components clients.
- 

Applicability

Use Decorator

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



Collaborations

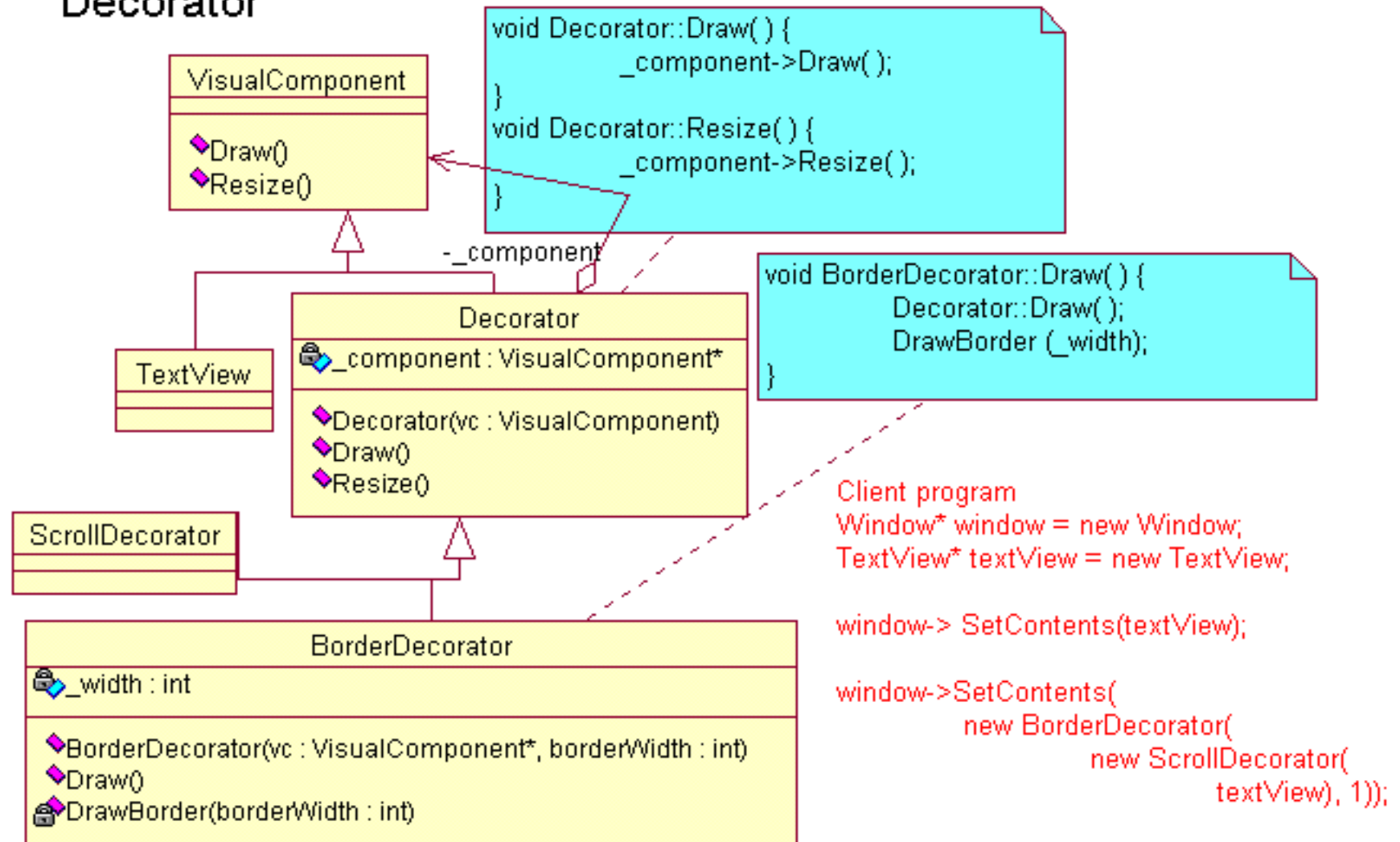
Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request.

Consequences

- ▶ **More flexibility than static inheritance:**
 - to add or delete responsibilities to objects at runtime simply by attaching and detaching them.
 - Providing different decorator classes for a specific component class lets you mix and match responsibilities.
- ▶ **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 features in a complex, customizable class, you can define simple class and add functionality incrementally with Decorator objects.
- ▶ **A decorator and its component aren't identical:**
 - A decorator acts as a transparent enclosure.
- ▶ **Lots of little objects :**
 - Decorator often results in system composed of lots of little objects that all look like, this may lead to maintenance nightmare and hard to learn.

Model

Decorator

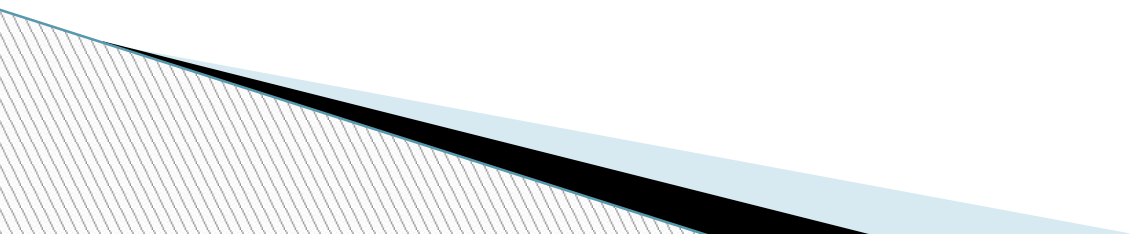


Related Patterns

Adapter: Adapter changes an object's interface, Decorator enhances an object's responsibilities. Decorator is thus more transparent to the client. As a consequence, Decorator supports recursive composition, which isn't possible with pure Adapters

Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface

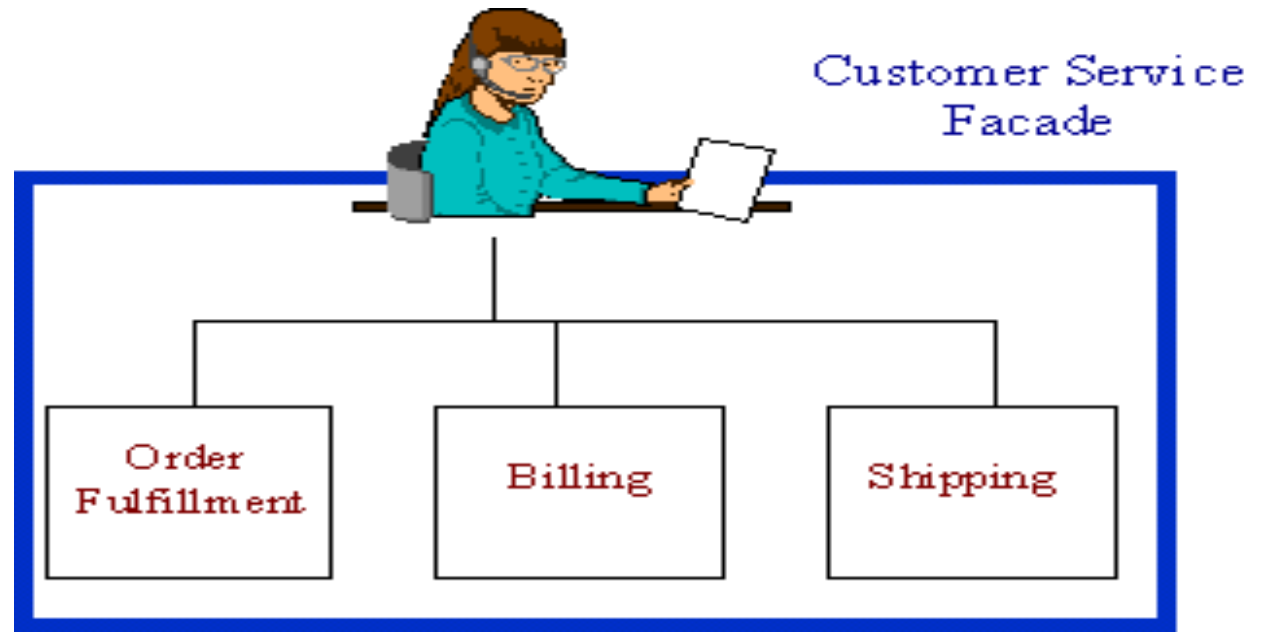
Composite: A decorator can be viewed as a degenerate composite with only one component. However, a decorator adds additional responsibilities—it isn't intended for object aggregation.



Facade

FACADE

Object Diagram for
Facade using Phone
Order Example

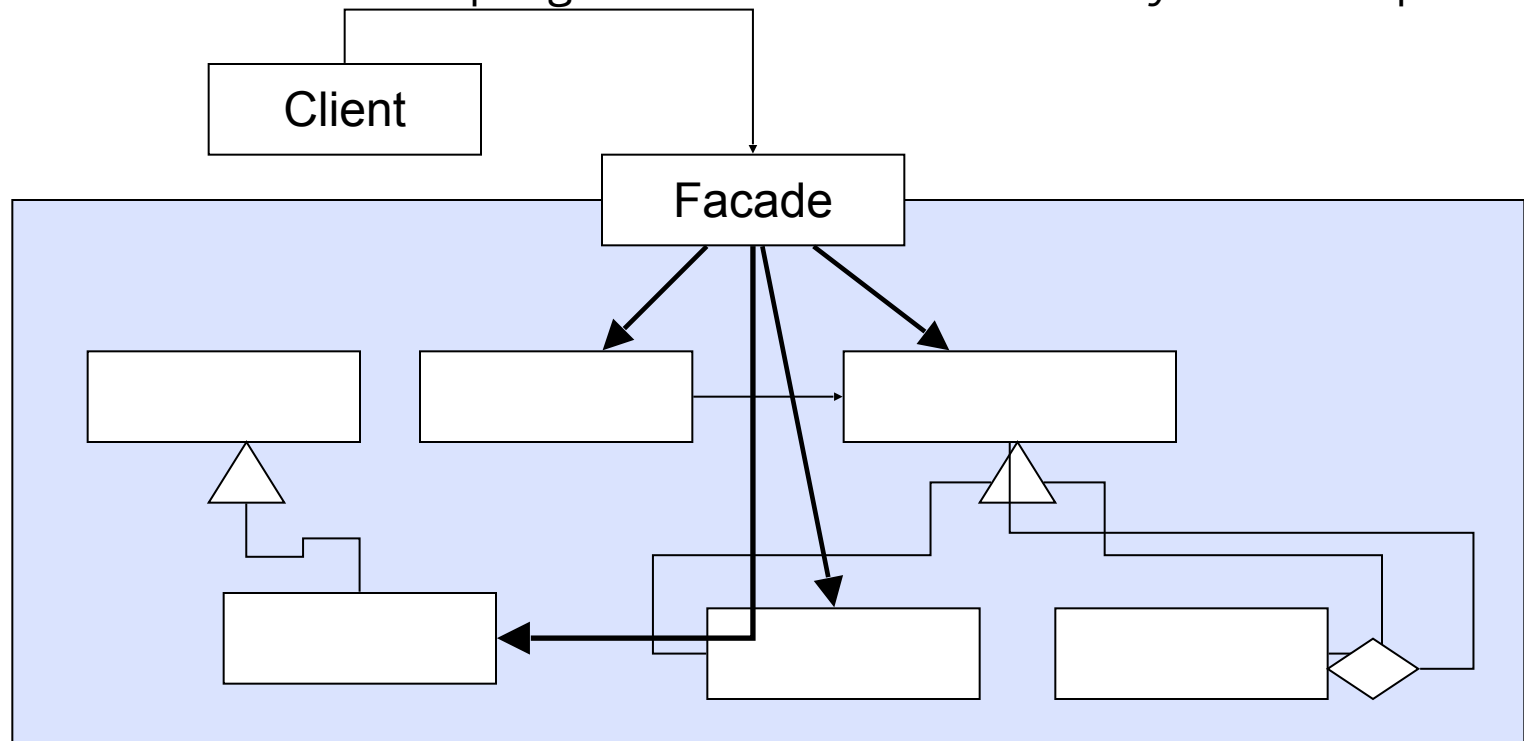


The *Facade* defines a unified, higher level interface to a subsystem, that makes it easier to use.

Consumers encounter a *Facade* when ordering from a catalog. The consumer calls one number and speaks with a customer service representative. The customer service representative acts as a *Facade*, providing an interface to the order fulfillment department, the billing department, and the shipping department.

Facade

- ▶ Provide unified interface to interfaces within a subsystem
- ▶ Shield clients from subsystem components
- ▶ Promote weak coupling between client and subsystem components



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.

Problem:

A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem

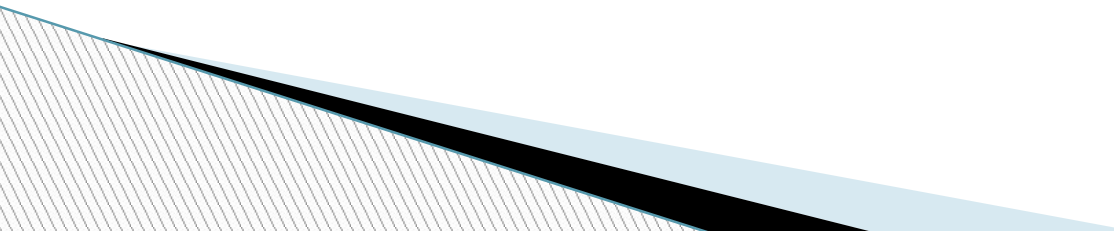


Illustration: Problem

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

But most clients of a compiler generally don't care about details like parsing and byte code generation; they merely want to compiler subsystem only complicate their task.

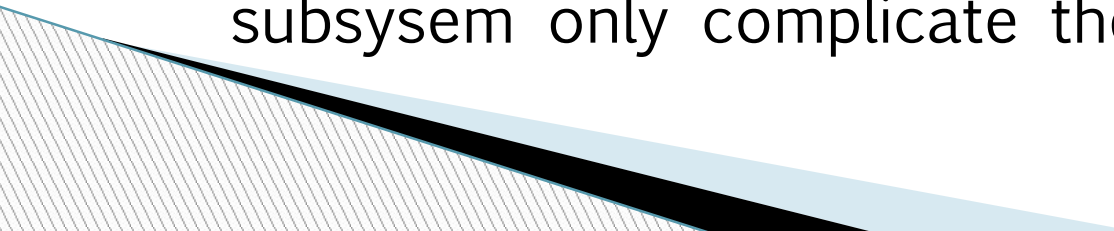
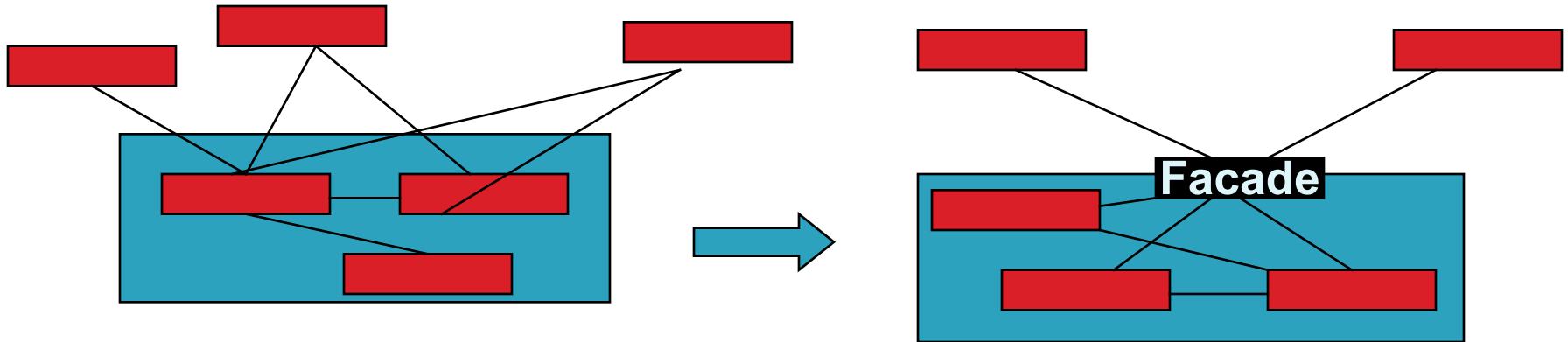


Illustration : solution

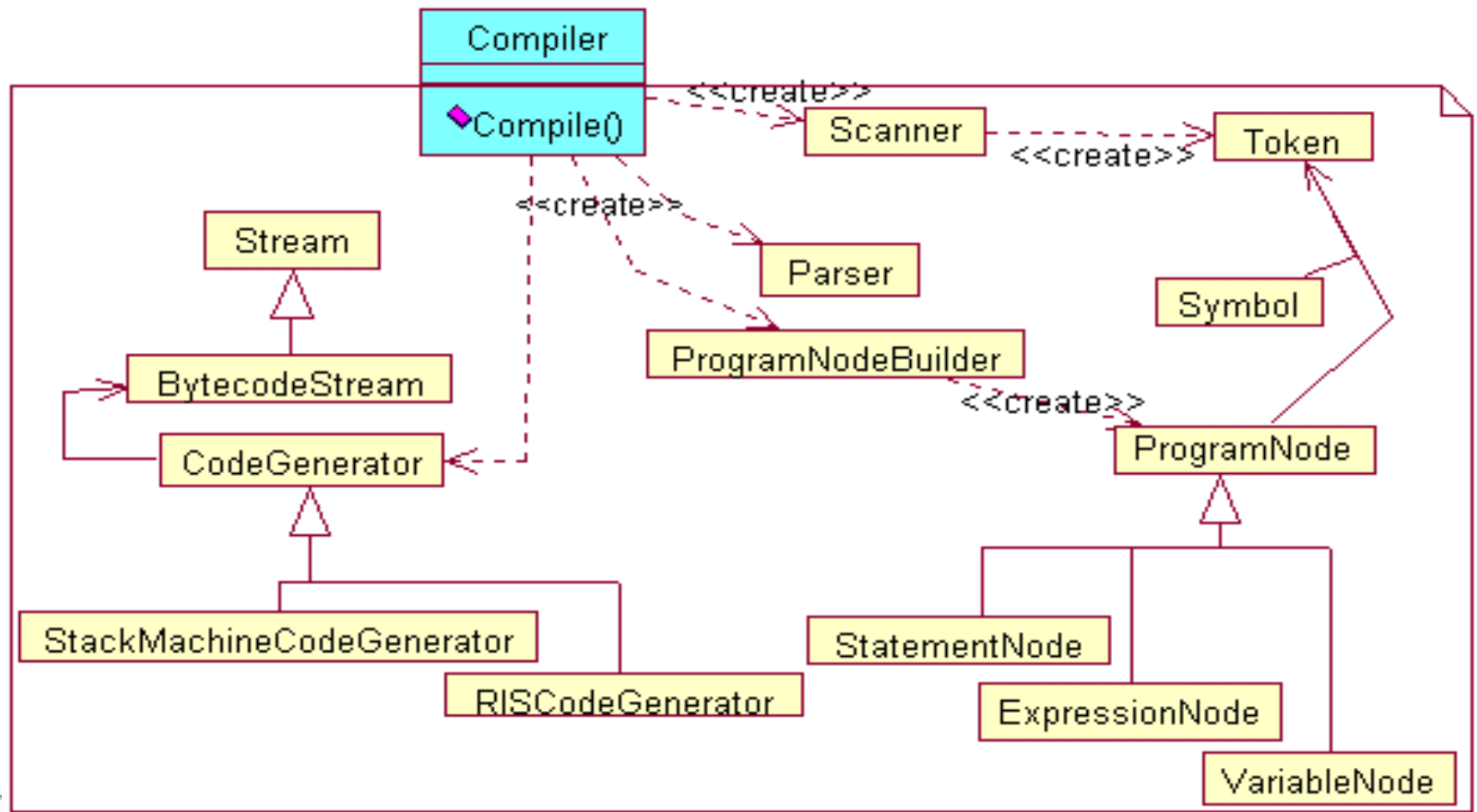


Compiler class in the above can act as a façade, which will provide higher-level interface that can shield clients from these classes, the compiler subsystem also includes a Compiler class. This class defines a unified interface to the compiler's functionality.

It offers client a single, simple interface to the compiler subsystem. It glues together the classes that implement compiler functionality without hiding them completely.

[Compiler façade makes easier for most programmers without hiding the lower-level functionality from the few that need it]

Illustration: solution



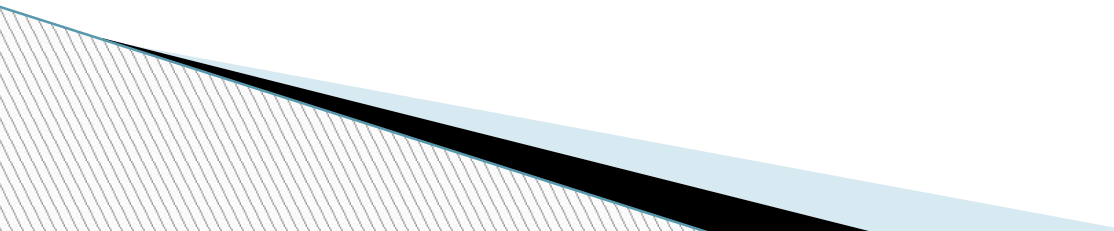
Applicability

Use the Façade pattern when

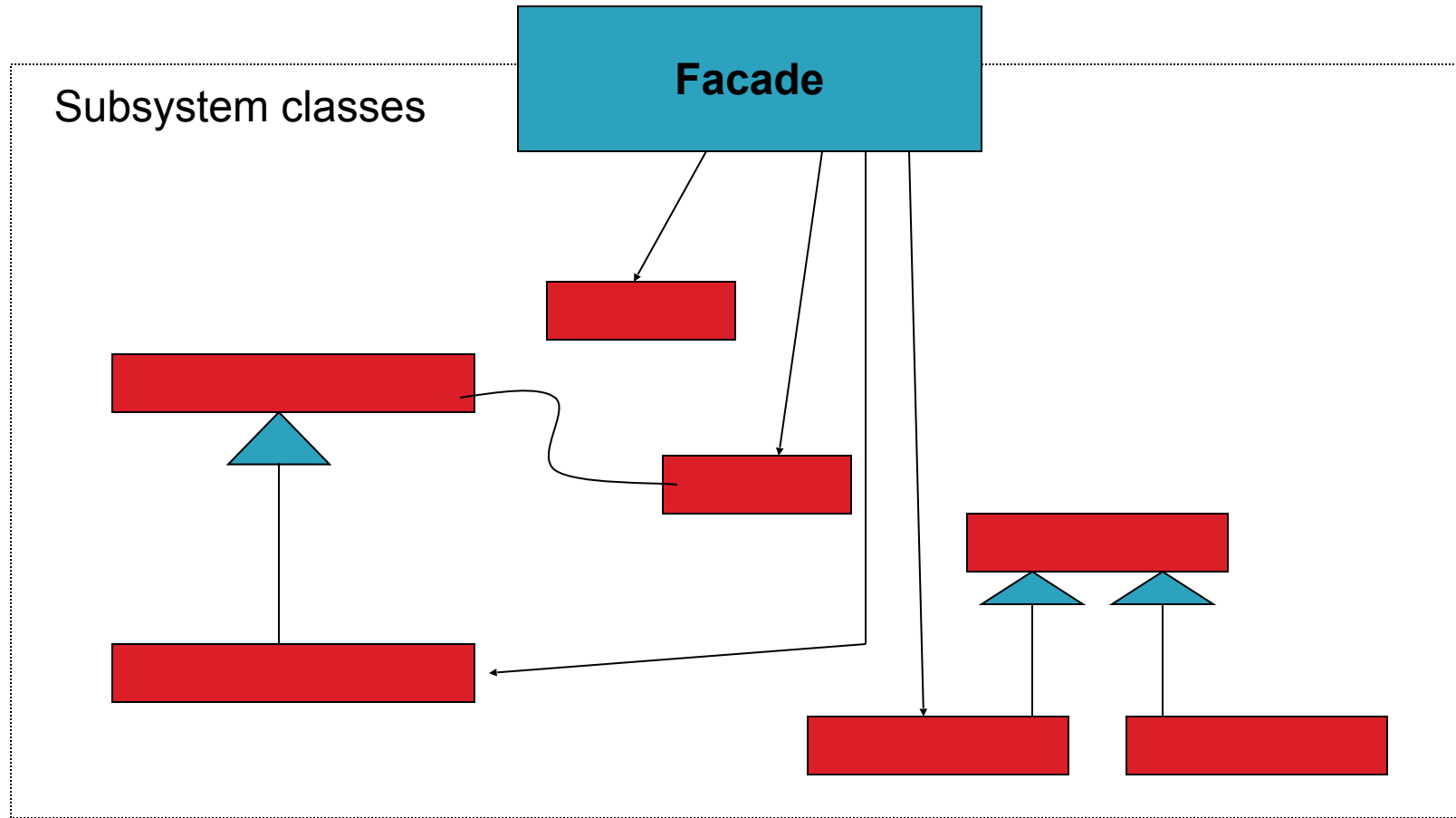
- ▶ **You want to provide a simple interface to a complex subsystem.**
 - To tackle the complexity of subsystem as they evolve
 - Most patterns, when applied, result in more and smaller classes. This makes the subsystem more reusable and easier to customize, but it also becomes harder to use for clients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

Applicability.....

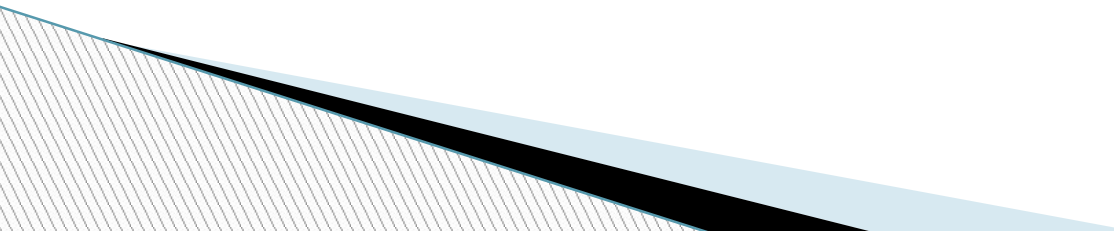
Use the Façade pattern when

- ▶ Façade to decouple the subsystem for clients and other subsystems, thereby promoting subsystem independence and portability which tackle the disadvantage of dependencies between clients and the implementation classes on an abstraction.
 - ▶ To layer your subsystem. Use façade 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.
- 

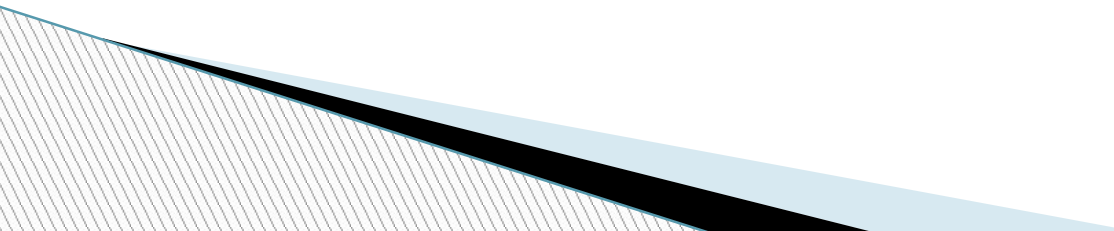
Structure



Collaborations

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

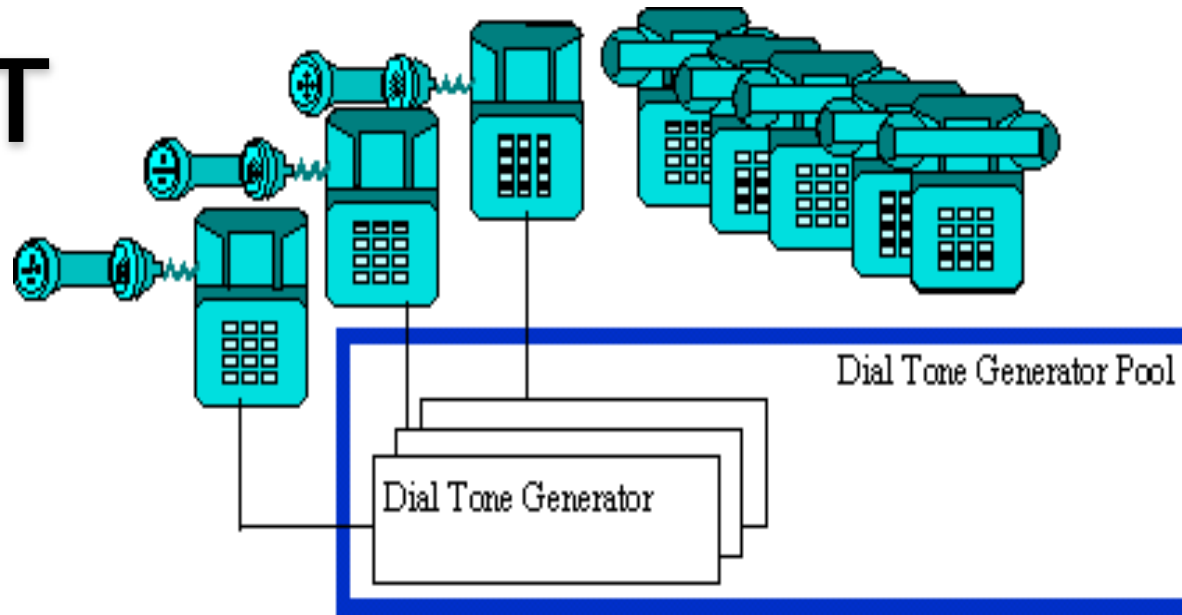
Consequences

- ▶ The Façade Patterns offers the following benefits:
 - ▶ 1. It shields client 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.
 - ▶ It doesn't prevent applications from using subsystem classes if they need to. Thus you can choose between ease of use and generality
- 

Flyweight

FLYWEIGHT

Dial Tone Generator
Example of *Flyweight*



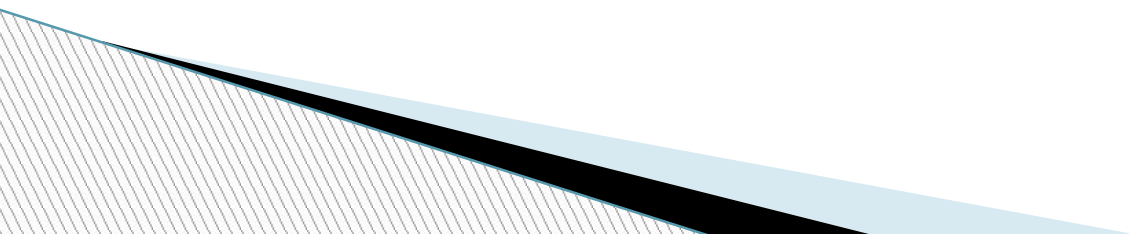
The *Flyweight* uses sharing to support large numbers of objects efficiently. The public switched telephone network is an example of a *Flyweight*. There are several resources such as dial tone generators, ringing generators, and digit receivers that must be shared between all subscribers. A subscriber is unaware of how many resources are in the pool when he or she lifts the hand set to make a call. All that matters to subscribers is that dial tone is provided, digits are received, and the call is completed.

Intent

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

Problem

- Designing objects down to the lowest levels of system "granularity" provides optimal flexibility, but can be unacceptably expensive in terms of performance and memory usage.

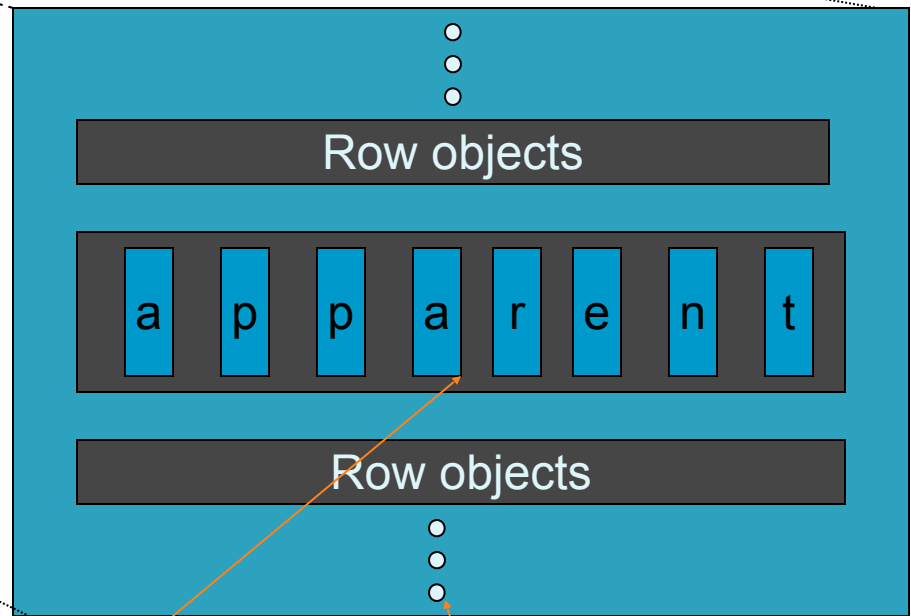


Illustration

Designing objects down to the lowest levels of system "granularity" provides optimal flexibility, but can be unacceptably expensive in terms

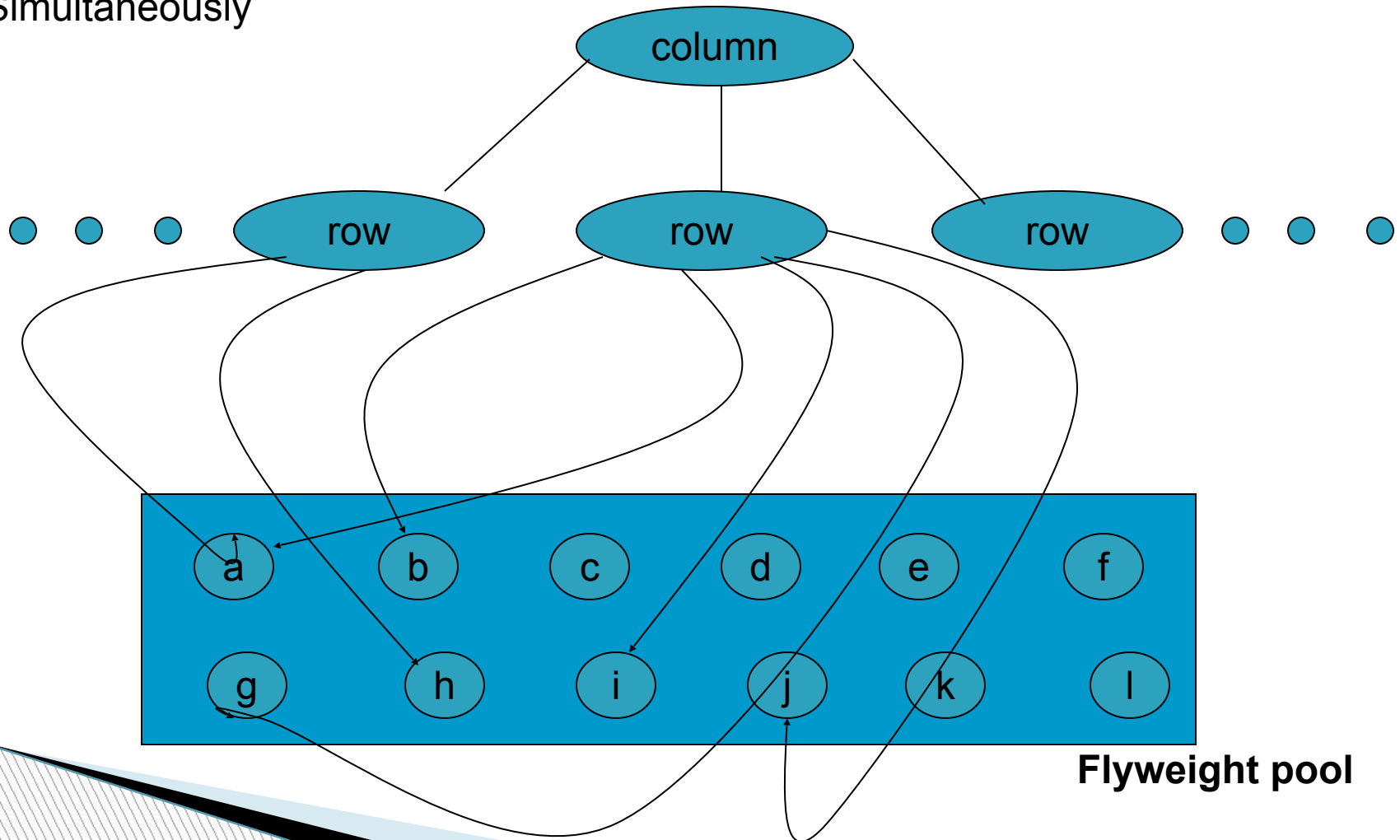
Character object

Column object

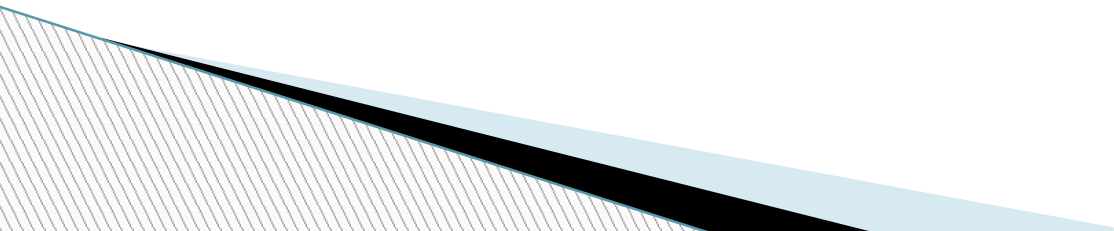


Illustration

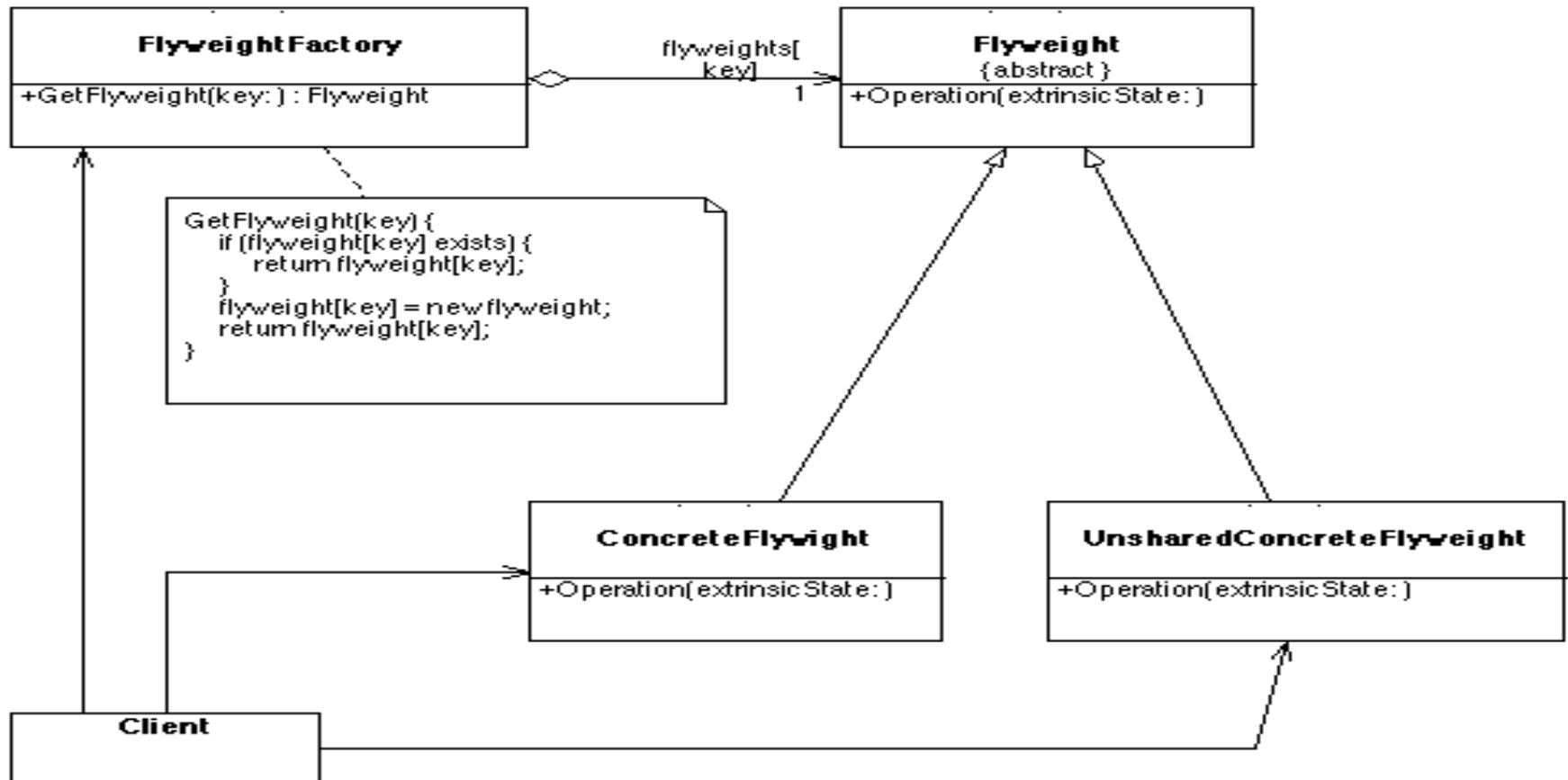
Flyweight is a shared object that can be used in multiple context
Simultaneously



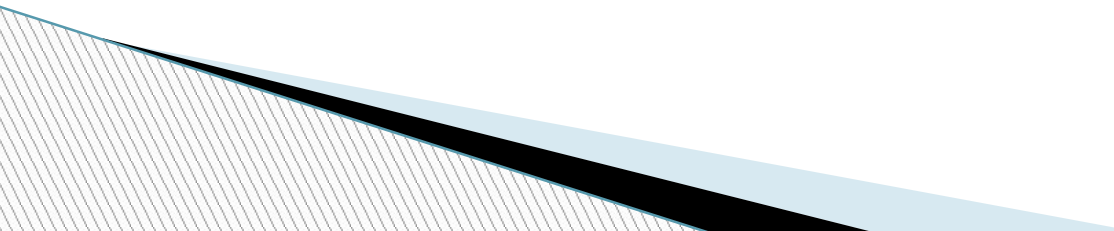
Applicability

- ▶ Apply in 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
- 

Structure



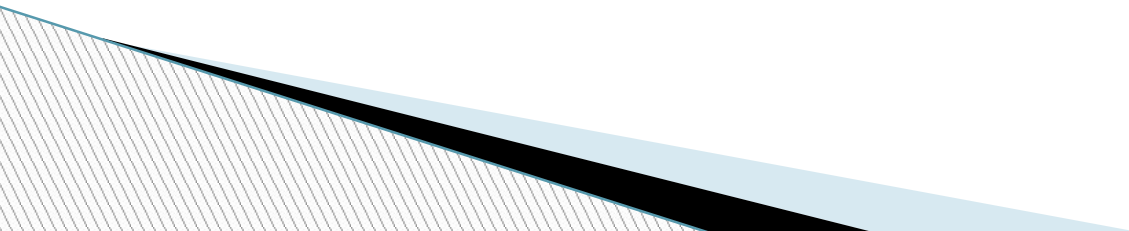
Collaborations

- ▶ State that a fly-weight 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. Client pass this state to the flyweight when they invoke its operations.
 - ▶ Client should not instantiate ConcreteFlyweights directly. Client must obtain ConcreteFlyweight objects exclusively from the FlyweightFactory object to ensure they are shared properly
- 

Consequences

Storage savings are functions of several factors:

- ▶ the reduction in the total number of instance that comes from sharing
- ▶ the amount of intrinsic state per object
- ▶ whether extrinsic state is computed or stored.



Thank You!

A decorative graphic element in the bottom-left corner of the slide, consisting of a series of parallel diagonal lines in a light beige color, set against a dark blue background.