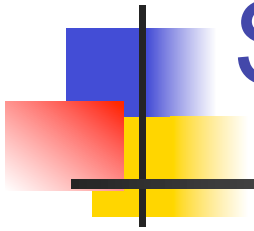
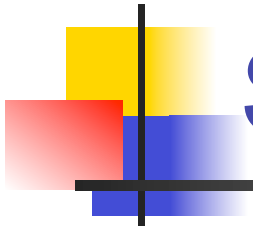
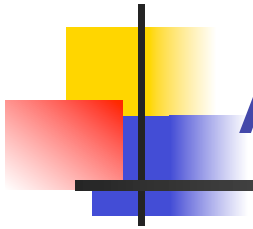


Structural Patterns



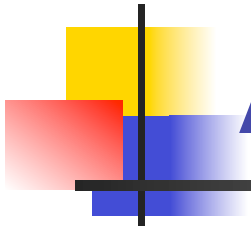
Structural Patterns

- Adapter (AD)
- Bridge (BR)
- Composite (CO)
- Decorator (DE)
- Facade (FA)
- Proxy (PR)



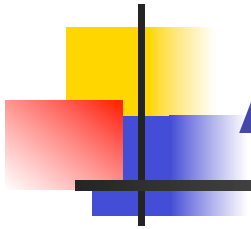
Adapter (AD)

- 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



AD Motivation (1)

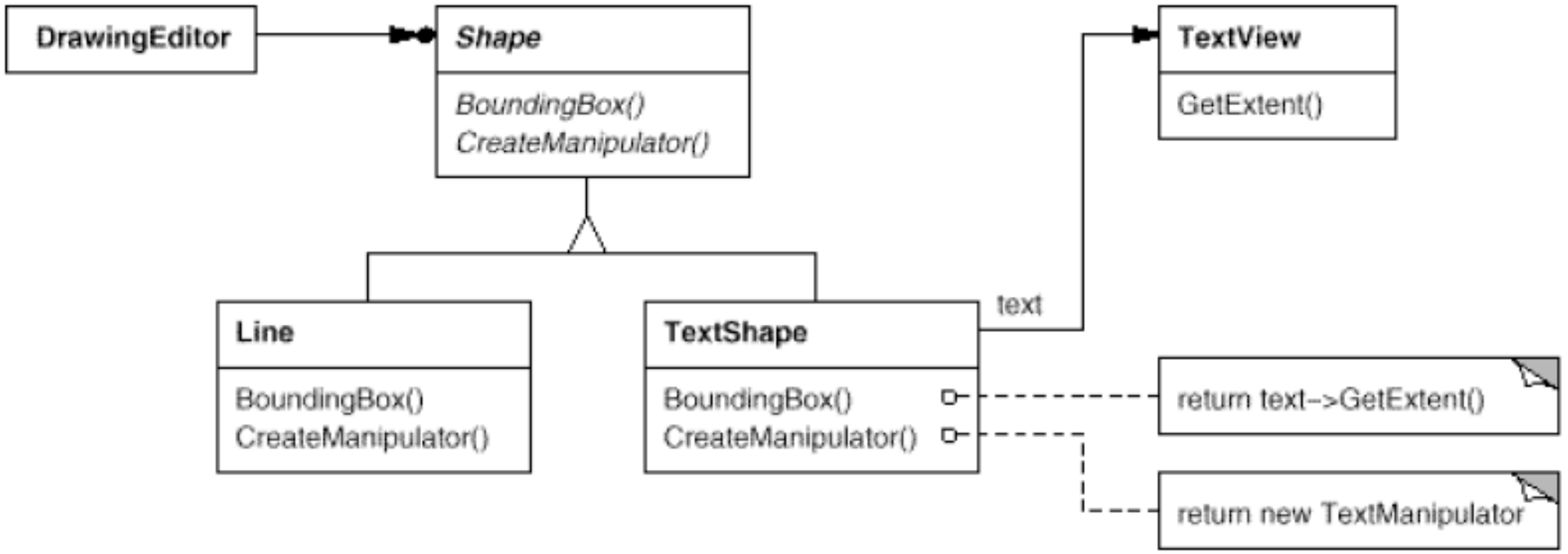
- If a class that's designed for reuse can’t be reused because its interface doesn't match the required domain-specific interface
- Consider a drawing editor that lets users draw and arrange lines, polygons, text, etc.
- The editor implements an abstract interface Shape which has an editable shape and can draw itself.
  - By subclassing Shape we implement:
    - a LineShape class for lines,
    - a PolygonShape class for polygons, etc.

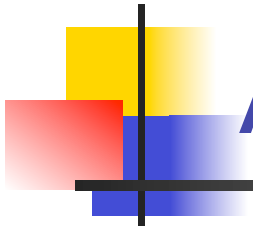


AD Motivation (2)

- But TextShapes are very complex w.r.t.. Lines!
- Suppose that there exists an off-the-shelf sophisticated toolkit for displaying and editing text.
  - Based on a TextView class
    - With an interface different from Shape
    - So we can't use TextView and Shape objects interchangeably.
- We could define TextShape so that it *adapts* the TextView interface to Shape's!!
  - This can be done by using Inheritance or Object-compostion

AD Motivation (3)



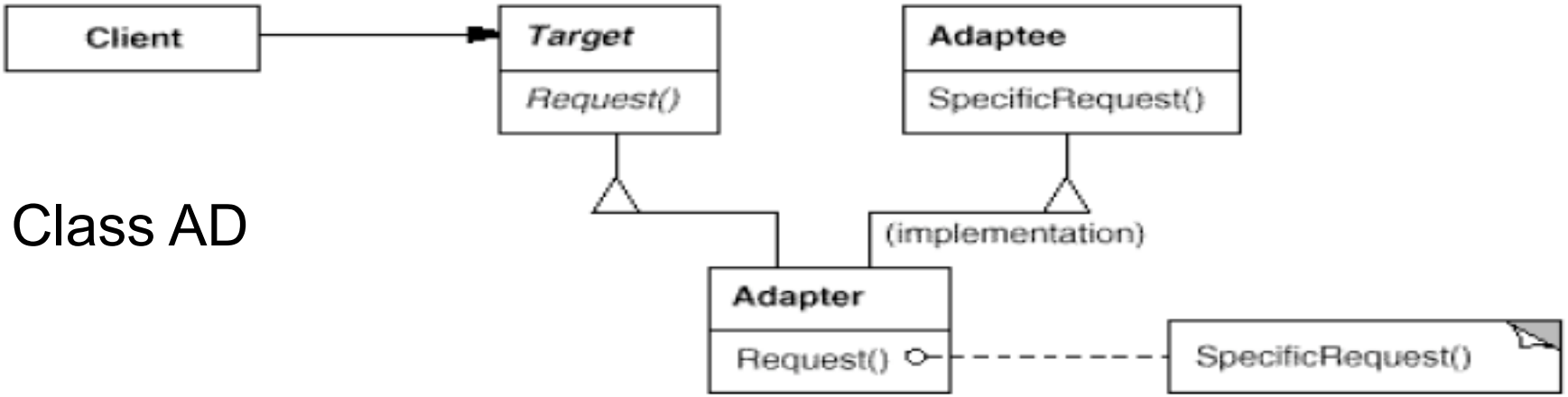


AD Applicability

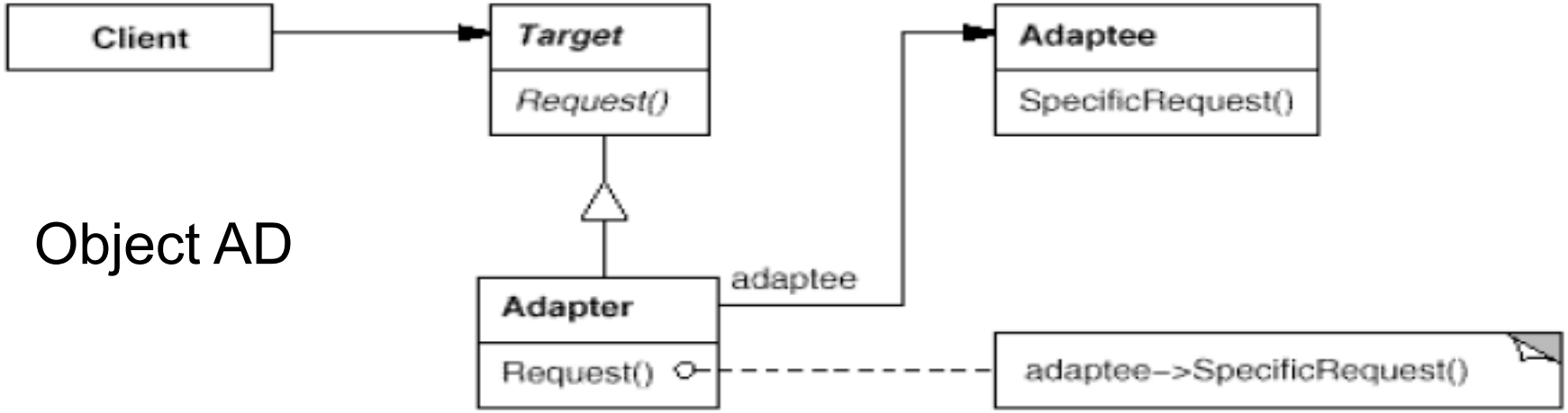
- Use AD when
  - you want to use an existing class, and its interface does not match the one you need.
  - you want to create a reusable class that cooperates classes that don't necessarily have compatible interfaces.
  - *(object adapter only)* you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one.

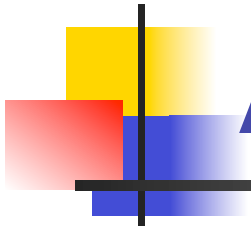
AD Structure

Class AD



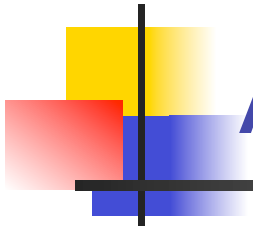
Object AD





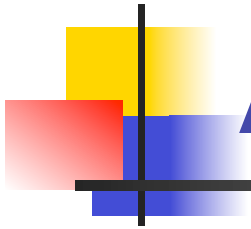
AD Participants

- Target (Shape)
  - defines the domain-specific interface that Client uses.
- Client (DrawingEditor)
  - collaborates with objects conforming to the Target interface.
- Adaptee (TextView)
  - defines an existing interface that needs adapting.
- Adapter (TextShape)
  - adapts the interface of Adaptee to the Target interface.



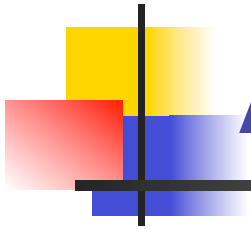
AD Collaboration

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



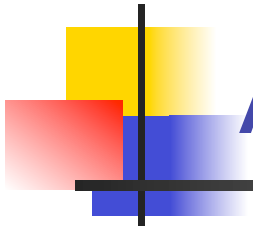
AD Consequences(1)

- About Class AD:
  - adapts Adaptee to Target by committing to a concrete Adapter class.
  - a class adapter won't work when we want to adapt a class and all its subclasses.
  - lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
  - introduces only one object, and no additional pointer indirection is needed to get to the adaptee.



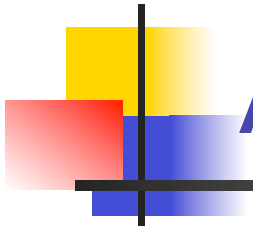
AD Consequences(2)

- About Object AD
  - 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.
  - makes it harder to override Adaptee behavior.
    - It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.



AD Consequences(3)

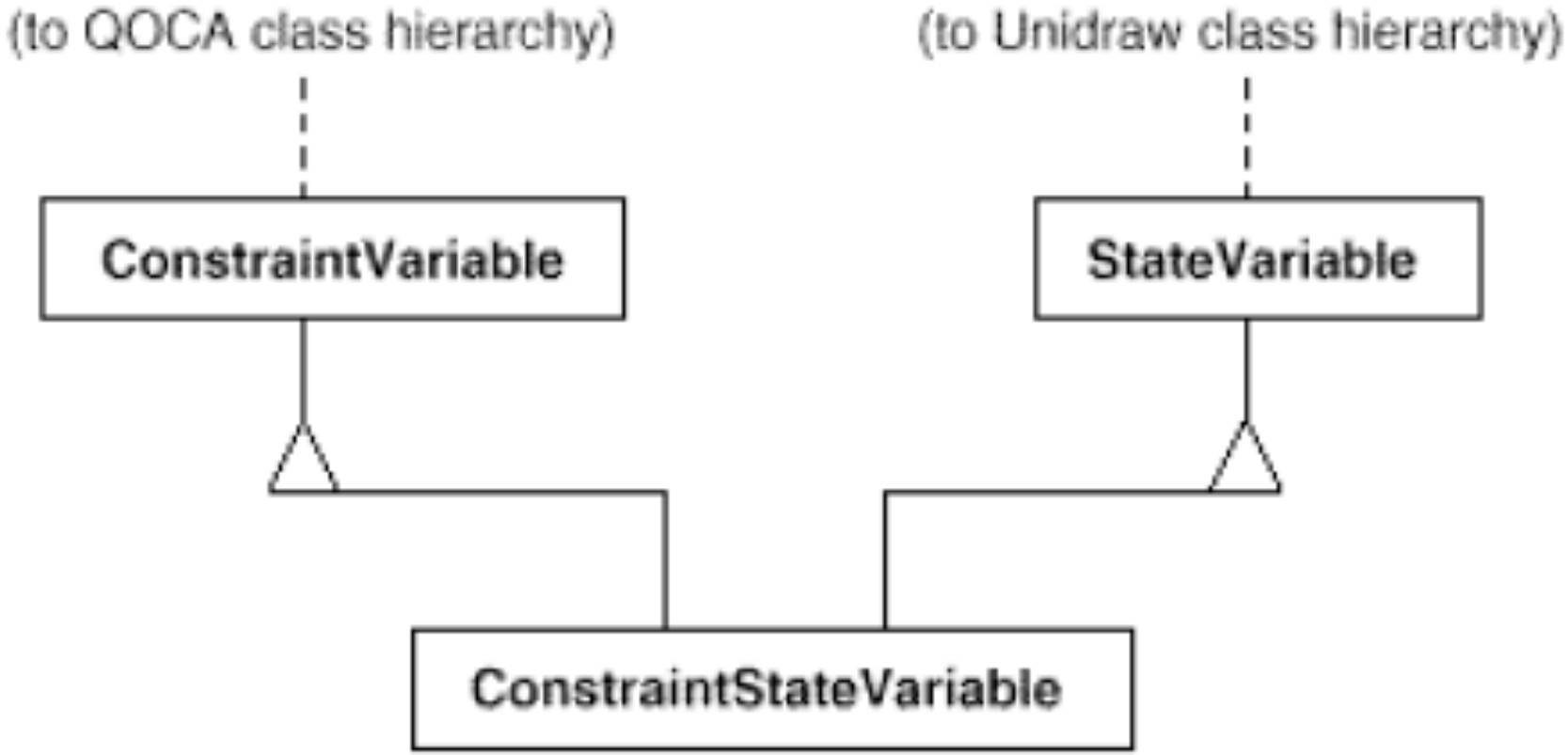
- How much adapting does Adapter do?
  - From renaming to adding features
  - depends on how similar the Target interface is to Adaptee's!!
- Pluggable adapters.
  - A class is more reusable when you minimize the assumptions other classes must make to use it.
  - By building interface adaptation into a class, you eliminate the assumption that other classes see the same interface.
  - Basically, to describe classes with built-in interface adaptation (see Implementation).

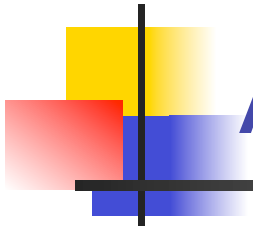


AD Consequences(4)

- Using two-way adapters to provide transparency.
  - adapters aren't transparent to all clients.
  - adapted object no longer conforms to the Adaptee interface
  - **Two-way adapters** can provide such transparency.

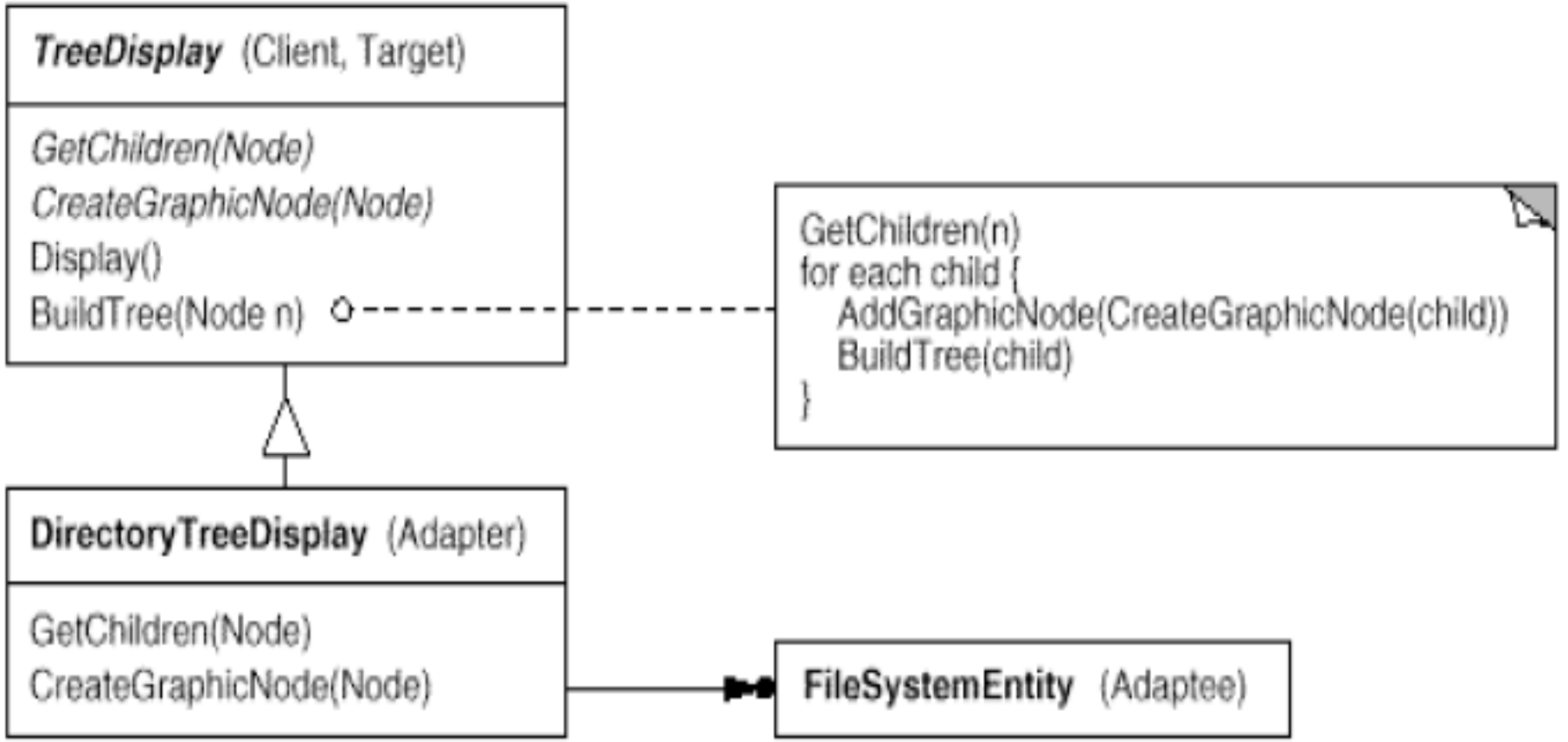
AD Consequences(4)

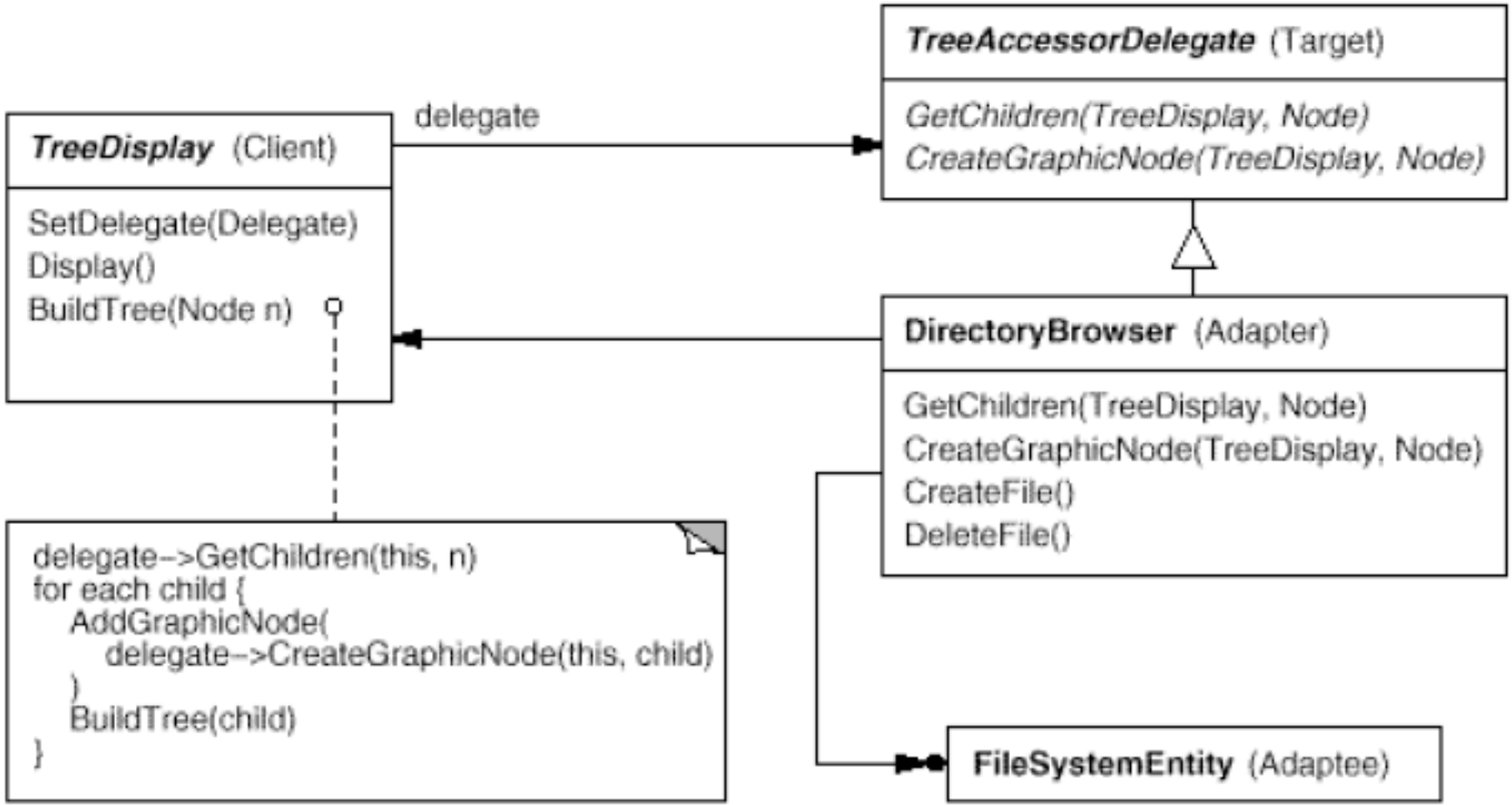


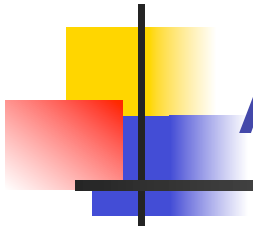


AD Implementation

- Implementing class adapters in C++.
  - In a C++ implementation of a class adapter, Adapter would inherit publicly from Target and privately from Adaptee.
  - Thus Adapter would be a subtype of Target but not of Adaptee.
- Pluggable adapters
  - Using abstract operations.
  - Using delegate objects.
  - Parameterized adapters (Smalltalk).


TreeDisplay (Client, Target)
GetChildren(Node)
CreateGraphicNode(Node)
Display()
BuildTree(Node n)
GetChildren(n)
for each child {
AddGraphicNode(CreateGraphicNode(child))
BuildTree(child)
}
DirectoryTreeDisplay (Adapter)
GetChildren(Node)
CreateGraphicNode(Node)
FileSystemEntity (Adaptee)


TreeAccessorDelegate (Target)
GetChildren(TreeDisplay, Node)
CreateGraphicNode(TreeDisplay, Node)
delegate
TreeDisplay (Client)
SetDelegate(Delegate)
Display()
BuildTree(Node n)
DirectoryBrowser (Adapter)
GetChildren(TreeDisplay, Node)
CreateGraphicNode(TreeDisplay, Node)
CreateFile()
DeleteFile()
delegate->GetChildren(this, n)
for each child {
AddGraphicNode(
delegate->CreateGraphicNode(this, child)
)
BuildTree(child)
}
FileSystemEntity (Adaptee)



AD Sample Code

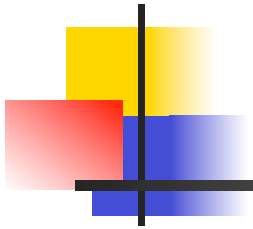
```
class Shape {
public:
    Shape();
    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual Manipulator* CreateManipulator() const;
};

class TextView {
public:
    TextView();
    void GetOrigin(Coord& x, Coord& y) const;
    void GetExtent(Coord& width, Coord& height) const;
    virtual bool IsEmpty() const;
};
```

```
class TextShape : public Shape, private TextView {
public:
    TextShape();

    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
};

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

    GetOrigin(bottom, left);
    GetExtent(width, height);

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

```
bool TextShape::IsEmpty () const {
    return TextView::IsEmpty();
}

Manipulator* TextShape::CreateManipulator () const {
    return new TextManipulator(this);
}
```

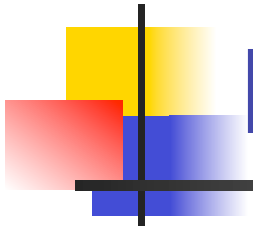
```
class TextShape : public Shape {
public:
    TextShape(TextView*);

    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
private:
    TextView* _text;
};

Manipulator* TextShape::CreateManipulator () const {
    return new TextManipulator(this);
}
```

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

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);
}

bool TextShape::IsEmpty () const {
    return _text->IsEmpty();
}
```



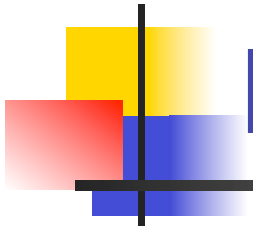
AD Known uses and Related Patterns

- Known Uses
  - ….many software
- Related Patterns
  - Bridge
  - Decorator
  - Proxy



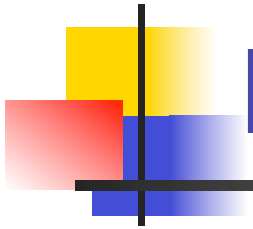
Bridge (BR)

- Intent
  - Decouple an abstraction from its implementation so that the two can vary independently.
- Also Known As
  - Handle/Body



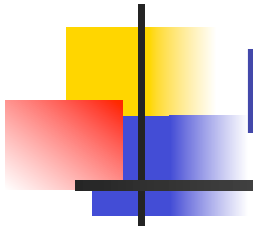
BR Motivation (1)

- When an abstraction can have one of several possible implementations, the usual way to accommodate them is to use inheritance.
  - an abstract class defines the interface to the abstraction
  - concrete subclasses implement it in different ways.
  - not flexible enough.
    - Inheritance binds an implementation to the abstraction permanently



BR Motivation (2)

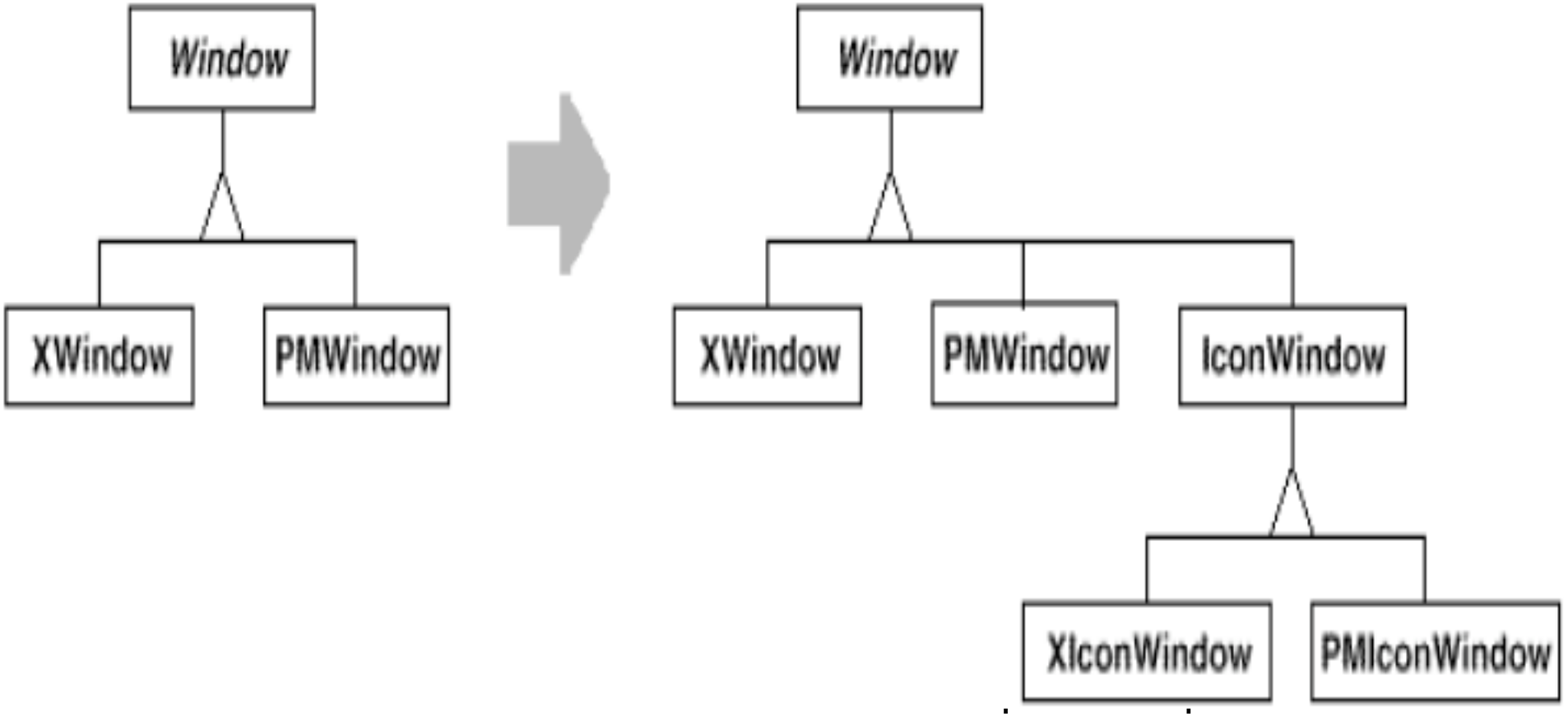
- Consider the implementation of a portable Window abstraction in a user interface toolkit.
  - We require it may support both the X Window System and IBM's Presentation Manager (PM)
  - This can be solved by using
    - Inheritance
    - Object composition



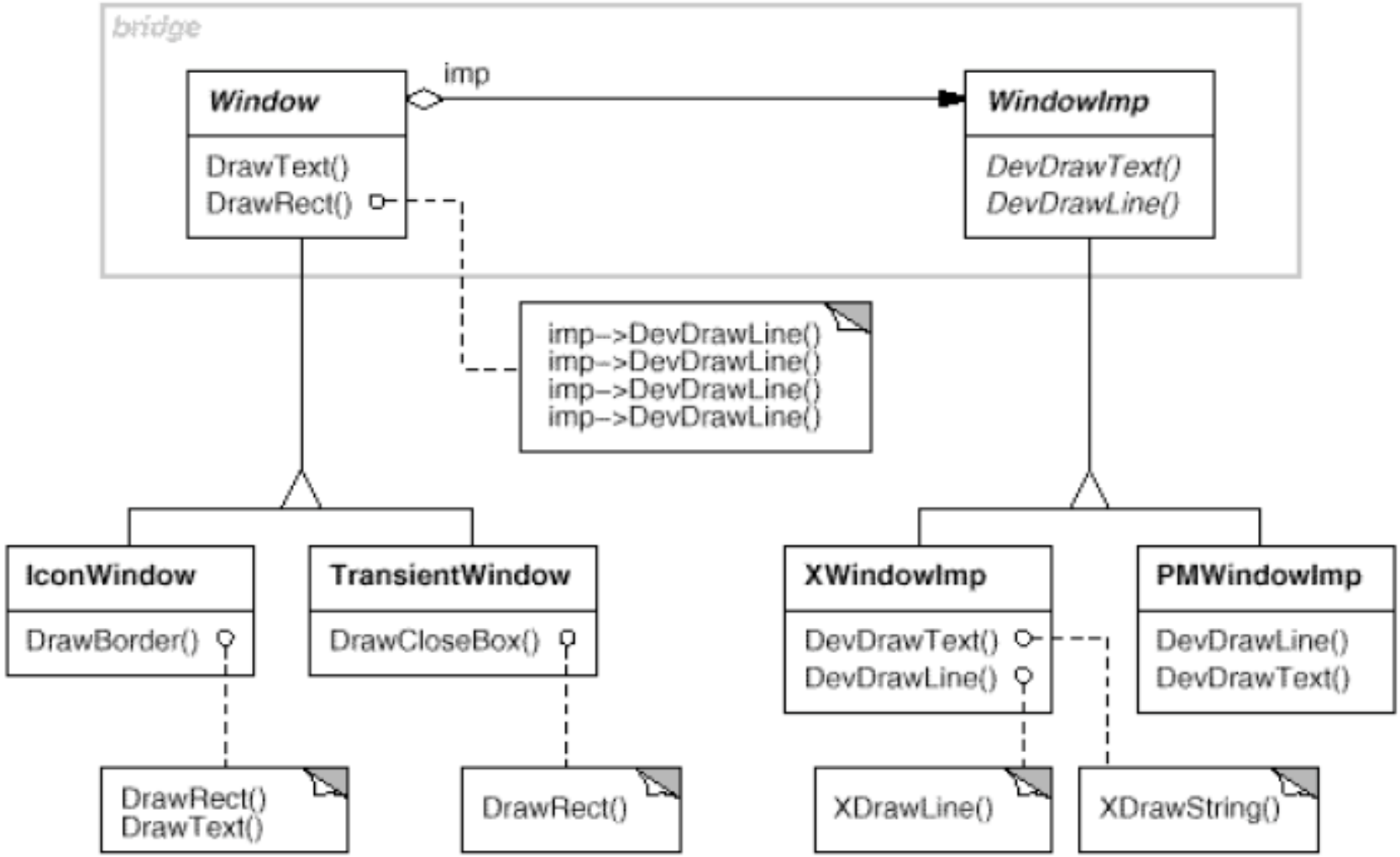
BR Motivation (3)

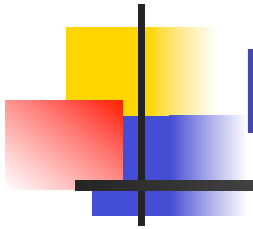
- If we use inheritance…
  - we define an abstract class Window and subclasses XWindow and PMWindow
  - But this approach has two drawbacks:
    - It's inconvenient to extend the Window abstraction to cover different kinds of windows or new platforms.
    - It makes client code platform-dependent.
      - Whenever a client creates a window, it instantiates a concrete class that has a specific implementation.

BR Motivation (3)



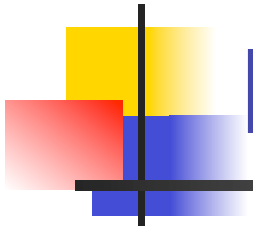
BR Motivation (4)





BR Applicability (1)

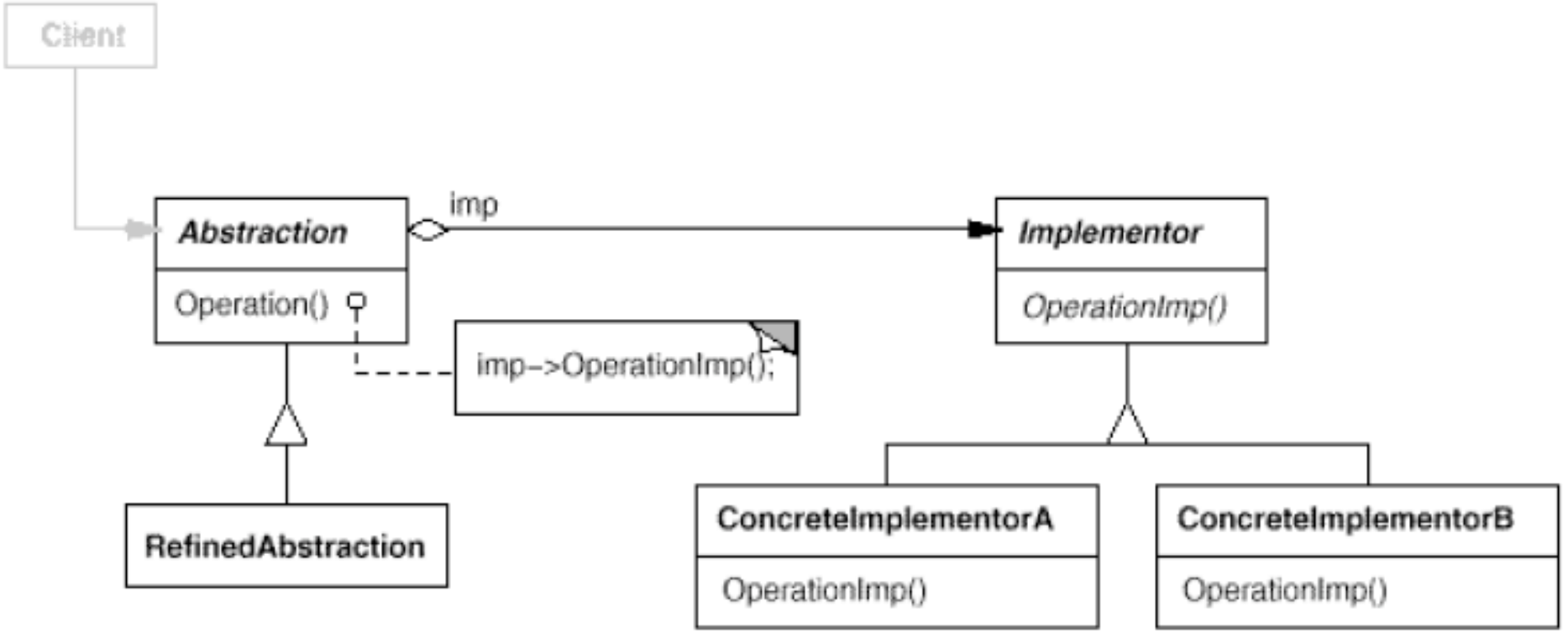
- Use the BR when:
  - you want to avoid a permanent binding between an abstraction and its implementation.
  - the implementation must be selected or switched at run-time.
  - both the abstractions and their implementations should be extensible by subclassing.
    - BR lets you combine the different abstractions and implementations and extend them independently.
  - The changes in the implementation shouldn’t require recompilation of the abstraction.

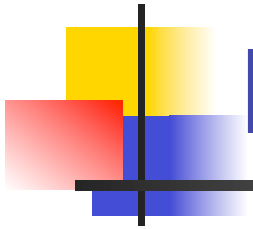


BR Applicability (2)

- Use the BR when:
  - (in C++) you want to hide the implementation of an abstraction completely from clients.
    - In C++ the representation of a class is visible in the class interface.
  - I you have a proliferation of classes as shown earlier in the first Motivation diagram.
  - I you want to share an implementation among multiple objects, and this fact should be hidden from the client.
    - A simple example is a string class in which multiple objects can share the same string representation.

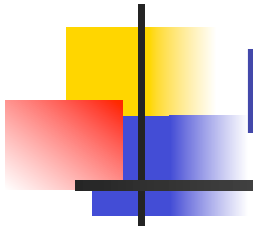
BR Structure





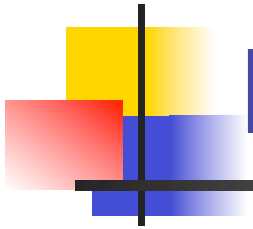
BR Participants

- Abstraction (Window)
  - defines the abstraction's interface.
  - maintains a reference to an Implementor.
- RefinedAbstraction (IconWindow)
  - Extends the interface defined by Abstraction.
- Implementor (WindowImp)
  - defines the interface for implementation classes.
    - Implementor and Abstraction may have different interface
      - Implementor (primitive); Abstraction (higher-level).
- ConcreteImplementor (XWindowImp, …)
  - implements the Implementor interface and defines its concrete implementation.



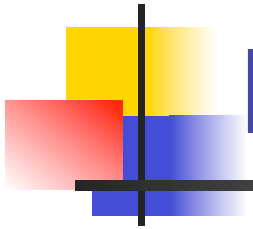
BR Collaborations

- Abstraction forwards client requests to its Implementor object.



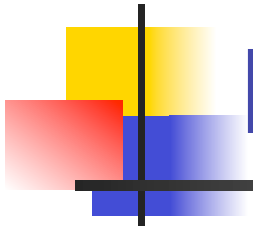
BR Consequences (1)

- Decoupling interface and implementation.
  - An implementation is not bound permanently to an interface.
  - The implementation of an abstraction can be configured/changed at run-time.
  - Eliminates compile-time dependencies on the implementation.
    - essential when you must ensure binary compatibility between different versions of a class library.
  - encourages layering that can lead to a better-structured system.



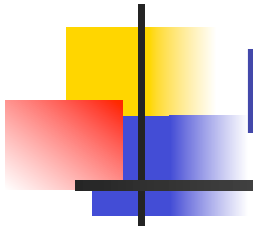
BR Consequences (2)

- Improved extensibility.
  - You can extend the Abstraction and Implementor hierarchies independently.
- Hiding implementation details from clients.
  - You can shield clients from implementation details, like the sharing of implementor objects and the accompanying reference count mechanism (if any).



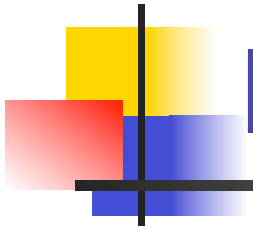
BR implementation (1)

- Only one Implementor.
  - a degenerate case
  - in situations where there's only one implementation, creating an abstract Implementor class isn't necessary.
  - this separation is still useful
    - change in the implementation shouldn't requirere compilation



BR implementation (2)

- Creating the right Implementor object.
  - instantiate one of them in its constructor
  - choose a default implementation initially and change it later according to usage.
  - delegate the decision to another object altogether
    - E.g. by using Abstract Factory and or Singleton
- Sharing implementors.
- Using multiple inheritance
  - in C++ to combine an interface with its implementation (inherit publicly from Abstraction and privately from a ConcreteImplementor)
    - it binds an implementation permanently to its interface.



BR Sample Code

- The following C++ code implements the Window/WindowImp example from the Motivation section.

```
class Window {
public:
    Window(View* contents);

    // requests handled by window
    virtual void DrawContents();

    virtual void Open();
    virtual void Close();
    virtual void Iconify();
    virtual void Deiconify();

    // requests forwarded to implementation
    virtual void SetOrigin(const Point& at);
    virtual void SetExtent(const Point& extent);
    virtual void Raise();
    virtual void Lower();

    virtual void DrawLine(const Point&, const Point&);
    virtual void DrawRect(const Point&, const Point&);
    virtual void DrawPolygon(const Point[], int n);
    virtual void DrawText(const char*, const Point&);

protected:
    WindowImp* GetWindowImp();
    View* GetView();

private:
    WindowImp* _imp;
    View* _contents; // the window's contents
};
```

```
class WindowImp {
public:
    virtual void ImpTop() = 0;
    virtual void ImpBottom() = 0;
    virtual void ImpSetExtent(const Point&) = 0;
    virtual void ImpSetOrigin(const Point&) = 0;

    virtual void DeviceRect(Coord, Coord, Coord, Coord) = 0;
    virtual void DeviceText(const char*, Coord, Coord) = 0;
    virtual void DeviceBitmap(const char*, Coord, Coord) = 0;
    // lots more functions for drawing on windows...
protected:
    WindowImp();
};
```

```
class ApplicationWindow : public Window {
public:
    // ...
    virtual void DrawContents();
};

void ApplicationWindow::DrawContents () {
    GetView()->DrawOn(this);
}
class IconWindow : public Window {
public:
    // ...
    virtual void DrawContents();
private:
    const char* _bitmapName;
};
```

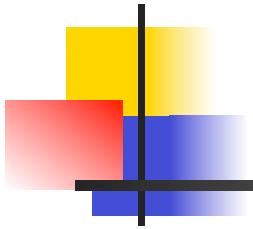
```
void IconWindow::DrawContents() {
    WindowImp* imp = GetWindowImp();
    if (imp != 0) {
        imp->DeviceBitmap(_bitmapName, 0.0, 0.0);
    }
}

void Window::DrawRect (const Point& p1, const Point& p2) {
    WindowImp* imp = GetWindowImp();
    imp->DeviceRect(p1.X(), p1.Y(), p2.X(), p2.Y());
}
```

```
class XWindowImp : public WindowImp {
public:
    XWindowImp();

    virtual void DeviceRect(Coord, Coord, Coord, Coord);
    // remainder of public interface...
private:
    // lots of X window system-specific state, including:
    Display* _dpy;
    Drawable _winid;  // window id
    GC _gc;           // window graphic context
};

class PMWindowImp : public WindowImp {
public:
    PMWindowImp();
    virtual void DeviceRect(Coord, Coord, Coord, Coord);

    // remainder of public interface...
private:
    // lots of PM window system-specific state, including:
    HPS _hps;
};
```

```
void XWindowImp::DeviceRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
    int x = round(min(x0, x1));
    int y = round(min(y0, y1));
    int w = round(abs(x0 - x1));
    int h = round(abs(y0 - y1));
    XDrawRectangle(_dpy, _winid, _gc, x, y, w, h);
}
```

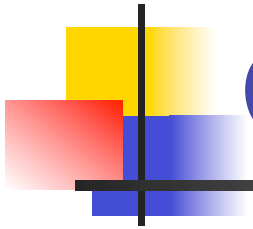
```
void PMWindowImp::DeviceRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
    Coord left = min(x0, x1);
    Coord right = max(x0, x1);
    Coord bottom = min(y0, y1);
    Coord top = max(y0, y1);

    PPOINTL point[4];

    point[0].x = left;    point[0].y = top;
    point[1].x = right;   point[1].y = top;
    point[2].x = right;   point[2].y = bottom;
    point[3].x = left;    point[3].y = bottom;

    if (
        (GpiBeginPath(_hps, 1L) == false) ||
        (GpiSetCurrentPosition(_hps, &point[3]) == false) ||
        (GpiPolyLine(_hps, 4L, point) == GPI_ERROR)  ||
        (GpiEndPath(_hps) == false)
    ) {
        // report error

    } else {
        GpiStrokePath(_hps, 1L, 0L);
    }
}
```

```
    WindowImp* Window::GetWindowImp () {
    if (_imp == 0) {
        _imp = WindowSystemFactory::Instance()->MakeWindowImp();
    }
    return _imp;
}
```



BR Known Uses and Related Patterns

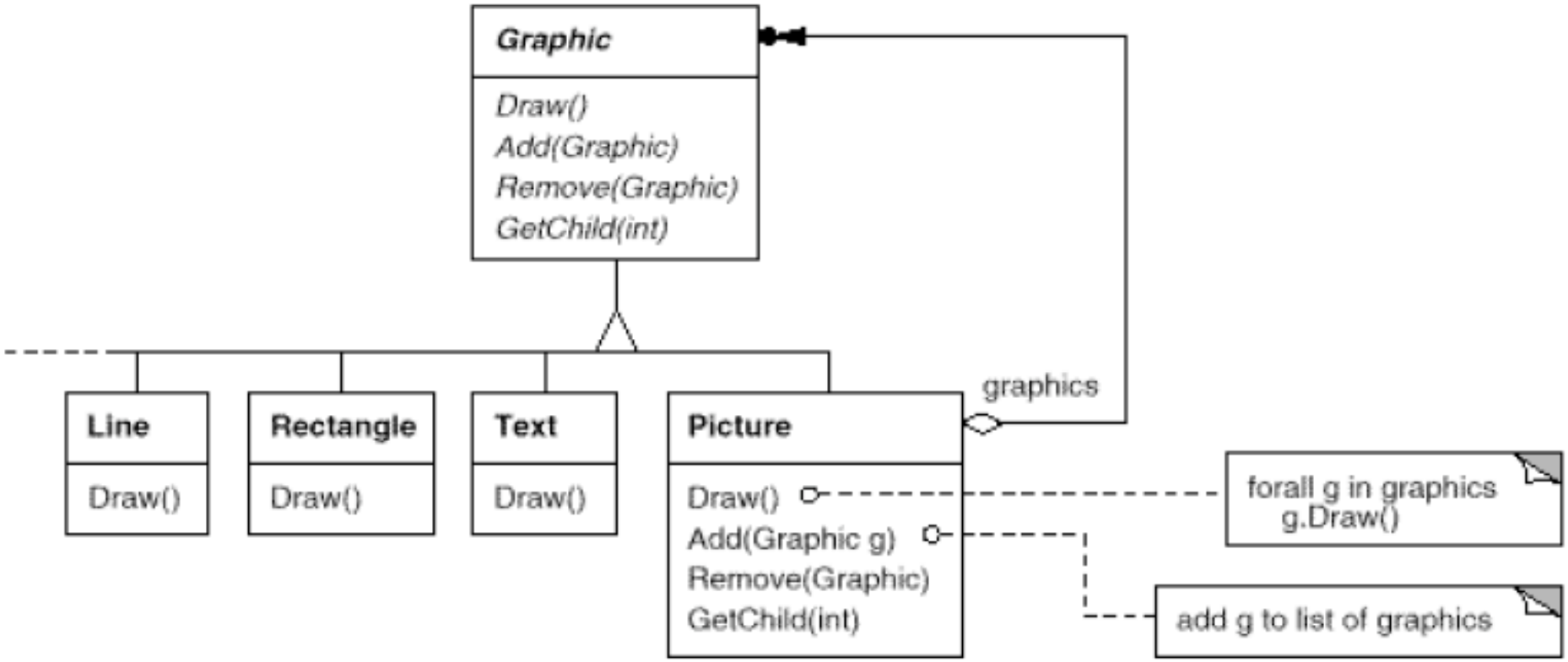
- Known Uses
  - …many applications
- Related Patters
  - Abstract Factory
  - Adapter



Composite (CO)

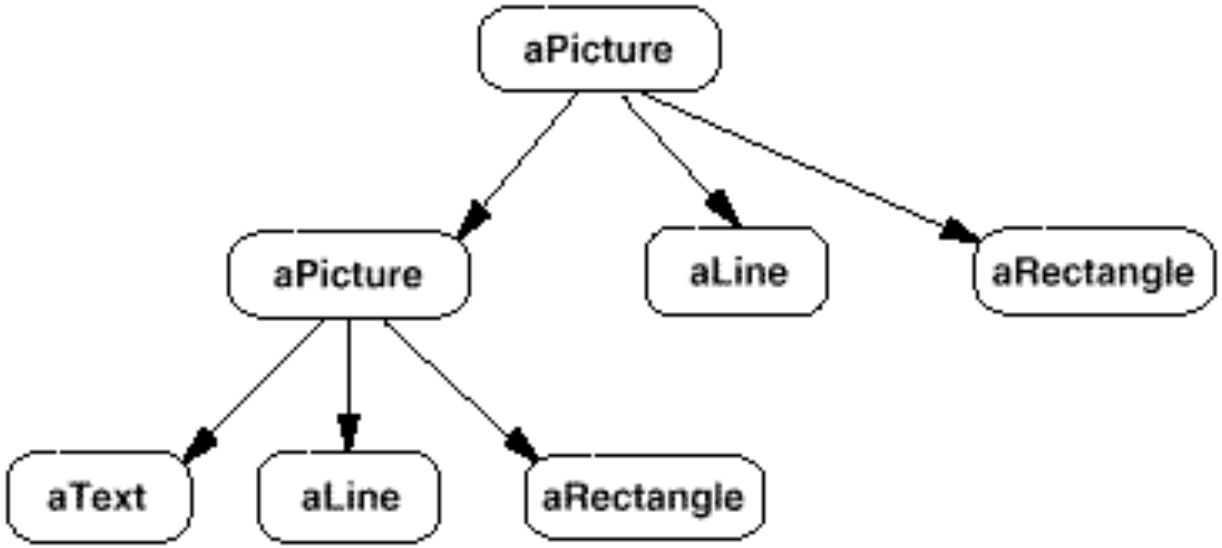
- Intent
  - Compose objects into tree structures to represent part-whole hierarchies.
  - Composite lets clients treat individual objects and compositions of objects uniformly.
- Motivation
  - Implement a drawing editor systems which lets users build complex diagrams out of simple components.
    - Components can be recursively grouped
    - Code that uses these classes must treat primitive and container objects differently
    - Having to distinguish these objects makes the application more complex.

CO Motivation (continued)

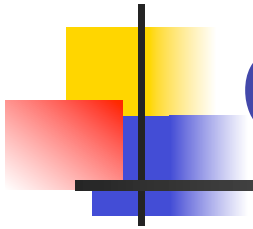


The Composite pattern is an abstract class that represents *both* primitives and their containers.
It also declares operations that all composite objects share, such as operations for accessing and managing its children.

CO Motivation (continued)



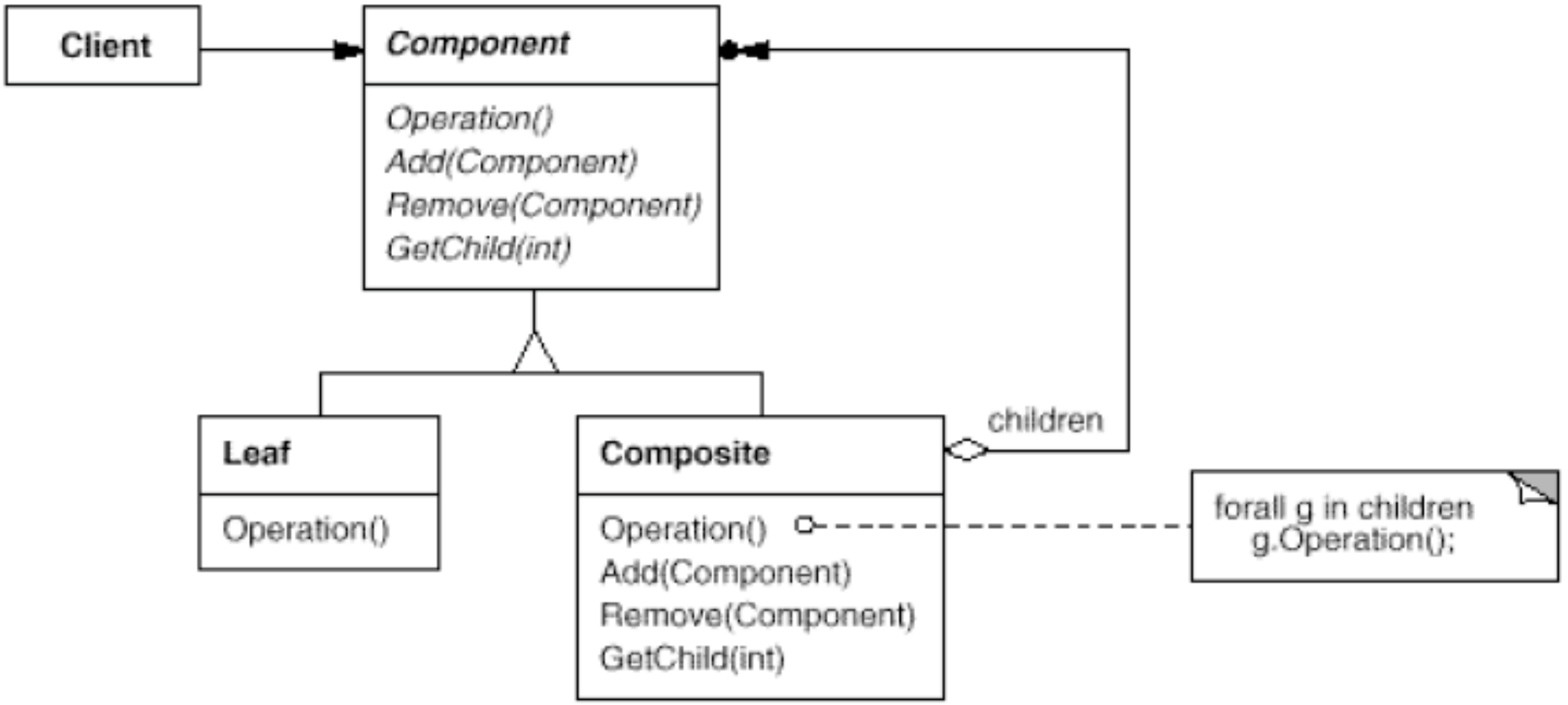
The Composite pattern is an abstract class that represents *both* primitives and their containers.
It also declares operations that all composite objects share, such as operations for accessing and managing its children.



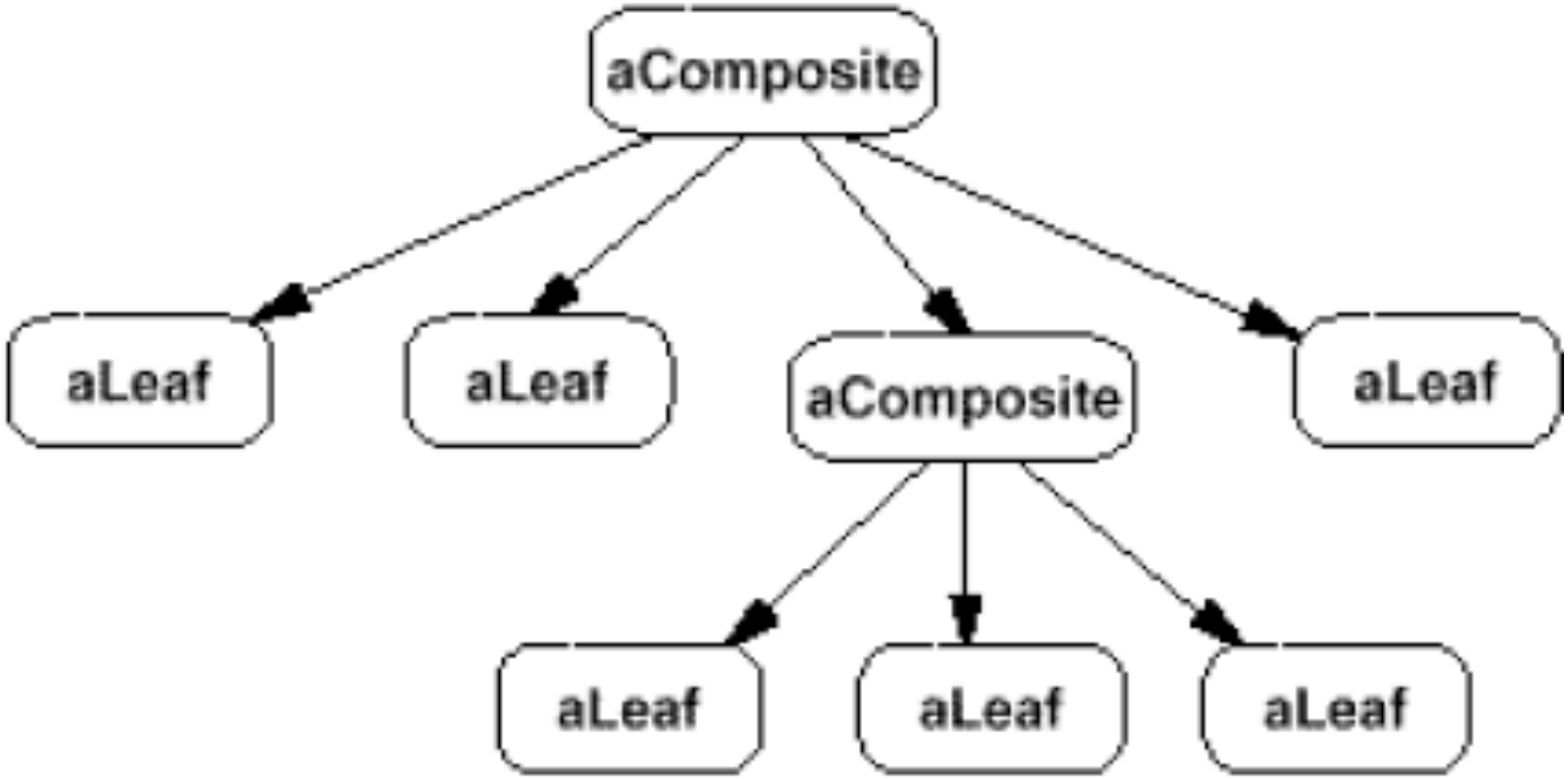
CO Applicability

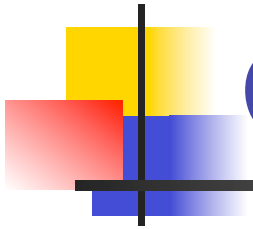
- Use the Composite pattern when
  - you want to represent part-whole hierarchies of objects.
  - you want clients to be able to ignore the difference between compositions of objects and individual objects.
  - clients will treat all objects in the composite structure uniformly.

CO Structure



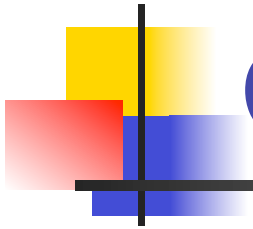
CO Structure





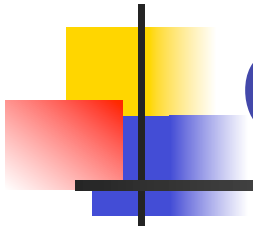
CO Participants (1)

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



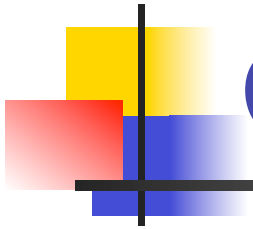
CO Participants (2)

- Leaf (Rectangle, Line, Text, etc.)
  - represents leaf objects in the composition.
  - A leaf has no children.
  - defines behavior for primitive objects
- Composite (Picture)
  - defines behavior for components having children.
  - stores child components.
  - implements child-related operations in the Component interface.
- Client
  - manipulates objects in the composition through the Component interface.



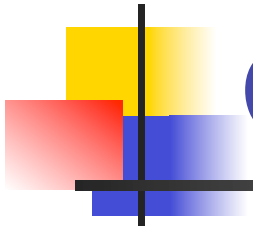
CO Collaborations

- 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.



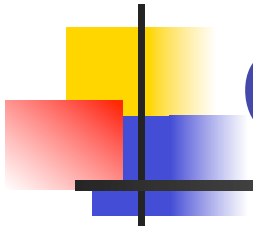
CO Consequences (1)

- defines class hierarchies consisting of primitive objects and composite objects.
  - Primitive objects can be composed into more complex objects and so on recursively.
  - Wherever client code expects a primitive object, it can also take a composite object.
- makes the client simple.
  - Clients can treat composite structures and individual objects uniformly.
    - This simplifies client code, because it avoids having to write tag-and-case-statement-style functions



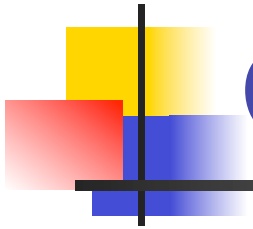
CO Consequences (1)

- makes it easier to add new kinds of components.
  - Newly defined Composite or Leaf subclasses work automatically with existing structures
  - Clients don't have to be changed for new Component classes.
- BUT it can make your design overly general.
  - it makes it harder to restrict the components of a composite.
  - you can't rely on the type system to enforce those constraints for you.
    - You'll have to use run-time checks instead.



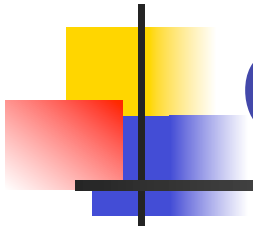
CO Implementation (1)

- Explicit parent references.
  - can simplify the traversal and management of a composite structure
    - Simplifies moving up the structure and deleting a component.
  - define the parent reference is in the Component
    - Leaf and Composite classes can inherit the reference and the operations
  - it's essential to maintain consistent the structure
    - change a component's parent only when it's being added or removed from a composite.



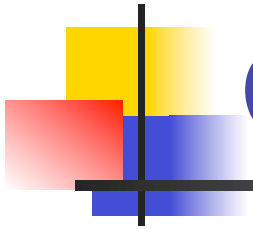
CO Implementation (2)

- Sharing components.
  - to reduce storage requirements.
  - But when a component can have no more than one parent, sharing components becomes difficult.
    - children could store multiple parents.
    - But that can lead to ambiguities (see Flyweight)



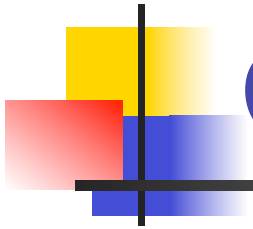
CO Implementation (3)

- Maximizing the Component interface.
  - to make clients unaware of the specific Leaf or Composite classes they're using.
    - The Component class should define as many common operations as possible.
    - The Component class usually provides default implementations for these operations, and Leaf and Composite subclasses will override them.
  - However, this goal will sometimes conflict with the principle of class hierarchy design that says a class should only define operations that are meaningful to its subclasses.
    - E.g. the interface for accessing children is a fundamental part of a Composite class but not necessarily Leaf classes.



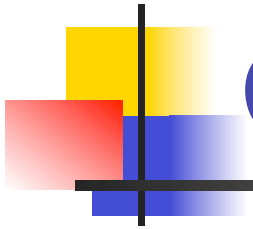
CO Implementation (4)

- Declaring the child management operations.
  - The Composite class (the root of the hierarchy) implements the Add and Remove operations
    - Is that correct?
  - safety or transparency?:
    - Defining it in Component (at the root)
      - gives you transparency;
        - you can treat all components uniformly.
      - It costs you safety;
        - clients may add and remove objects from leaves.
    - Defining it in Composite
      - gives you safety, no add/rem in the leaves
      - you lose transparency; leaves and composites have different interfaces.



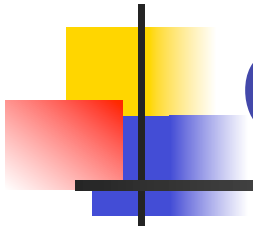
CO Implementation (5)

- Child ordering.
  - Many designs specify an ordering on the children of Composite. (e.g. parse trees)
  - The Iterator pattern can guide you in this.
- Caching to improve performance.
  - If you need to traverse or search compositions frequently, the Composite class can cache traversal or search information about its children.
  - The Composite can cache actual results or just information that lets it short-circuit the traversal or search.



CO Implementation (6)

- Who should delete components?
  - In languages without garbage collection, it's usually best to make a Composite responsible for deleting its children when it's destroyed.
  - An exception to this rule is when Leaf objects are immutable and thus can be shared.
- What's the best data structure for storing components?
  - may use a variety of data structures
    - E.g. linked lists, trees, arrays, and hash tables.
  - The choice of data structure depends (as always) on efficiency.



CO Sample Code

Equipment such as computers and stereo components are often organized into part-whole or containment

hierarchies.

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

    const char* Name() { return _name; }

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

    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator* CreateIterator();
protected:
    Equipment(const char*);
private:
    const char* _name;
};
```

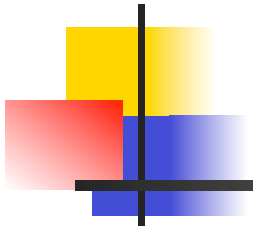
```
class FloppyDisk : public Equipment {
public:
    FloppyDisk(const char*);
    virtual ~FloppyDisk();

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

```
class CompositeEquipment : public Equipment {
public:
    virtual ~CompositeEquipment();

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

    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator* CreateIterator();

protected:
    CompositeEquipment(const char*);
private:
    List _equipment;
};
```

```
Currency CompositeEquipment::NetPrice () {
    Iterator* i = CreateIterator();
    Currency total = 0;

    for (i->First(); !i->IsDone(); i->Next()) {
        total += i->CurrentItem()->NetPrice();
    }
    delete i;
    return total;
}
```

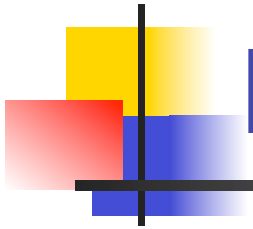
```
class Chassis : public CompositeEquipment {
public:
    Chassis(const char*);
    virtual ~Chassis();

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

```
Cabinet* cabinet = new Cabinet("PC Cabinet");
Chassis* chassis = new Chassis("PC Chassis");

cabinet->Add(chassis);

Bus* bus = new Bus("MCA Bus");
bus->Add(new Card("16Mbs Token Ring"));

chassis->Add(bus);
chassis->Add(new FloppyDisk("3.5in Floppy"));

cout << "The net price is " << chassis->NetPrice() << endl;
```



CO Known Uses and Related Patterns

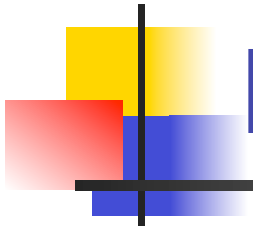
- Known Uses
  - …many applications
- Related Patterns
  - Iterator
  - Flyweigth
  - Decorator
  - Visitor



Decorator (DE)

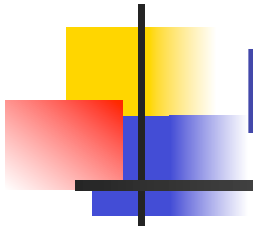
- Intent
  - Attach additional responsibilities to an object dynamically.
  - Decorators provide a flexible alternative to subclassing for extending functionality.
- Also Known As
  - Wrapper

DE Motivation



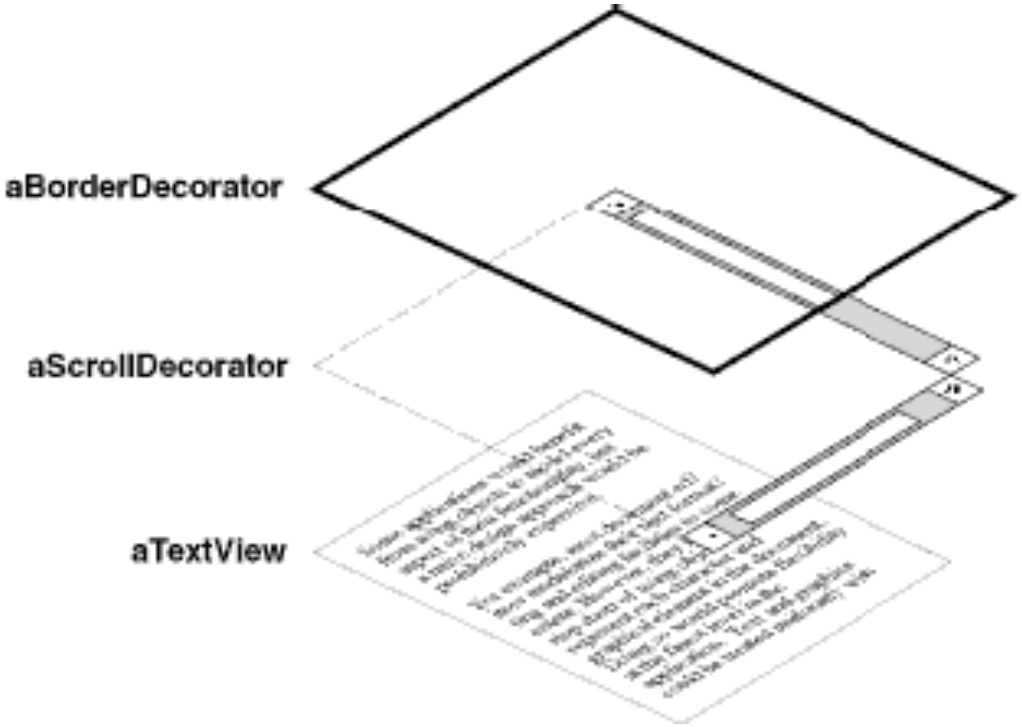
DE Motivation

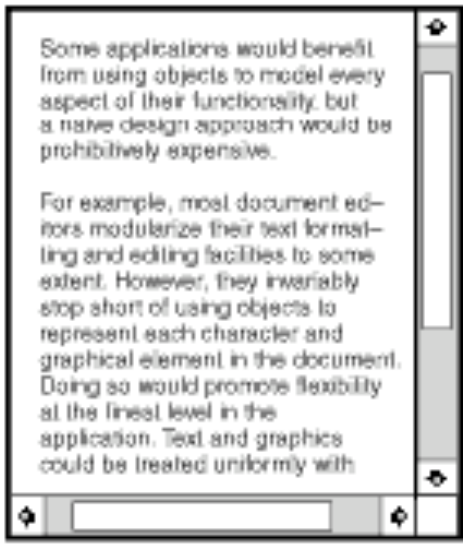
- Sometimes we want to add responsibilities to individual objects, not to an entire class.
  - A graphical user interface toolkit should let you add properties like borders or behaviours like scrolling to any user interface component.
- One way to add responsibilities is with inheritance.
  - Inheriting a border from another class puts a border around every subclass instance.
  - This is inflexible
    - the choice of border is made statically.
    - A client can't control how and when to decorate the component with a border.

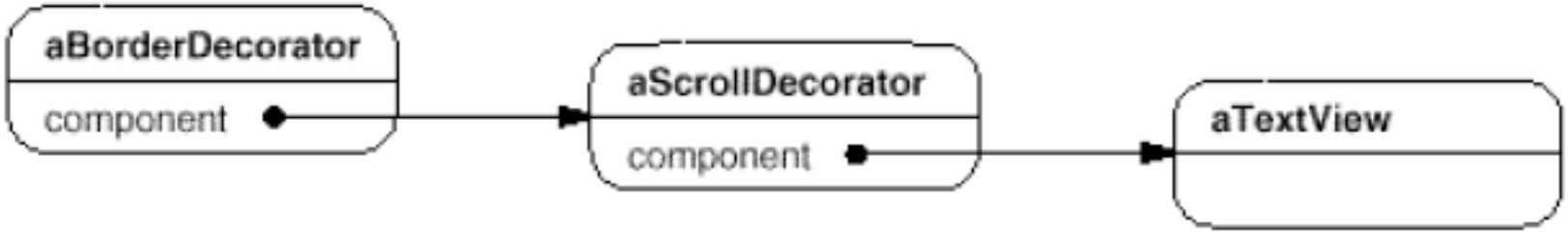


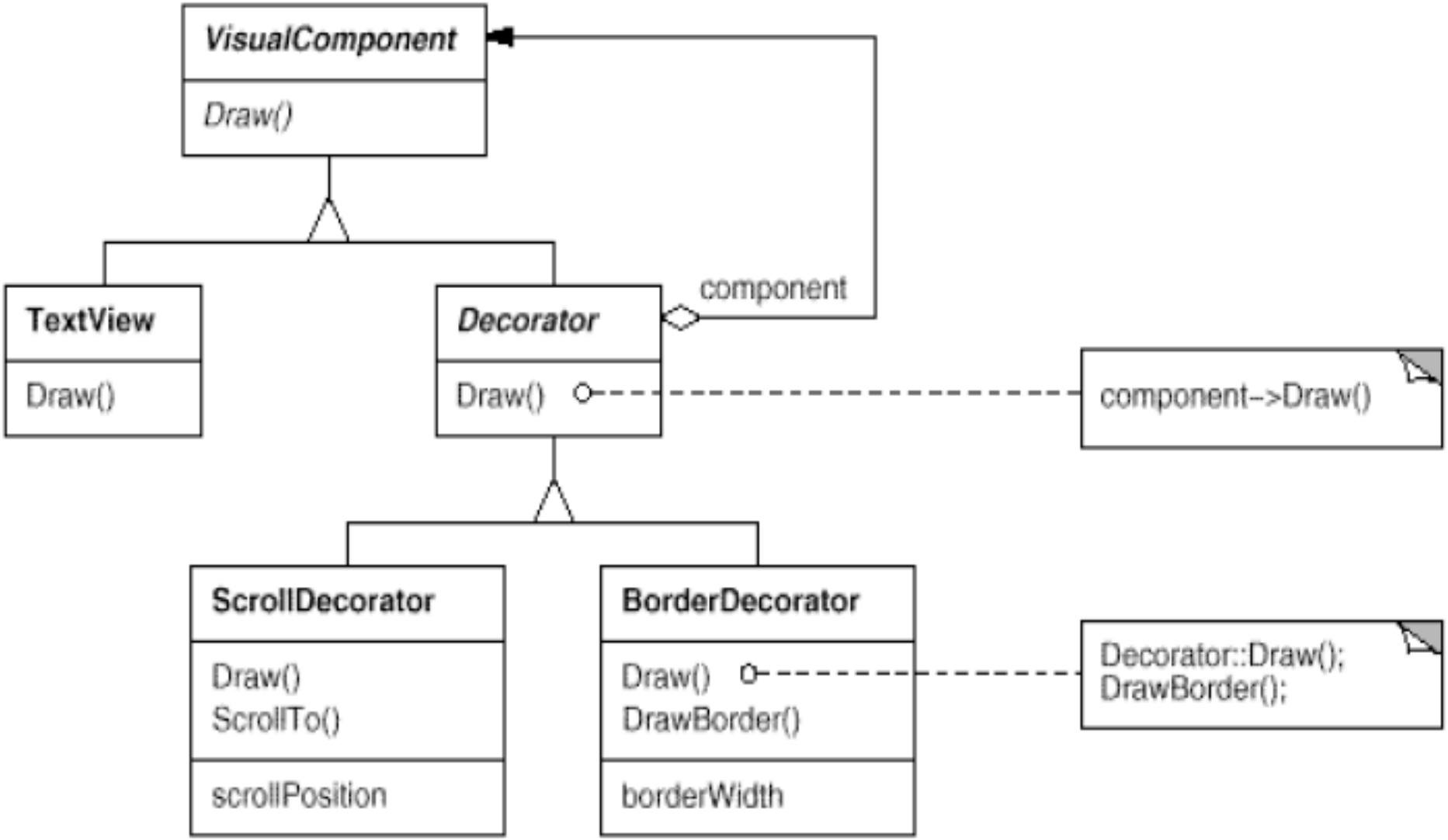
Decorator Motivation

- 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
  - It 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
  - allowing an unlimited number of added responsibilities.


aBorderDecorator
aScrollDecorator
aTextView

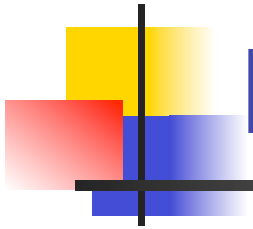

Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.
For example, most document editors modularize their text formatting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the finest level in the application. Text and graphics could be treated uniformly with


aBorderDecorator
component
aScrollDecorator
component
aTextView



DE lets decorators appear anywhere a VisualComponent can.

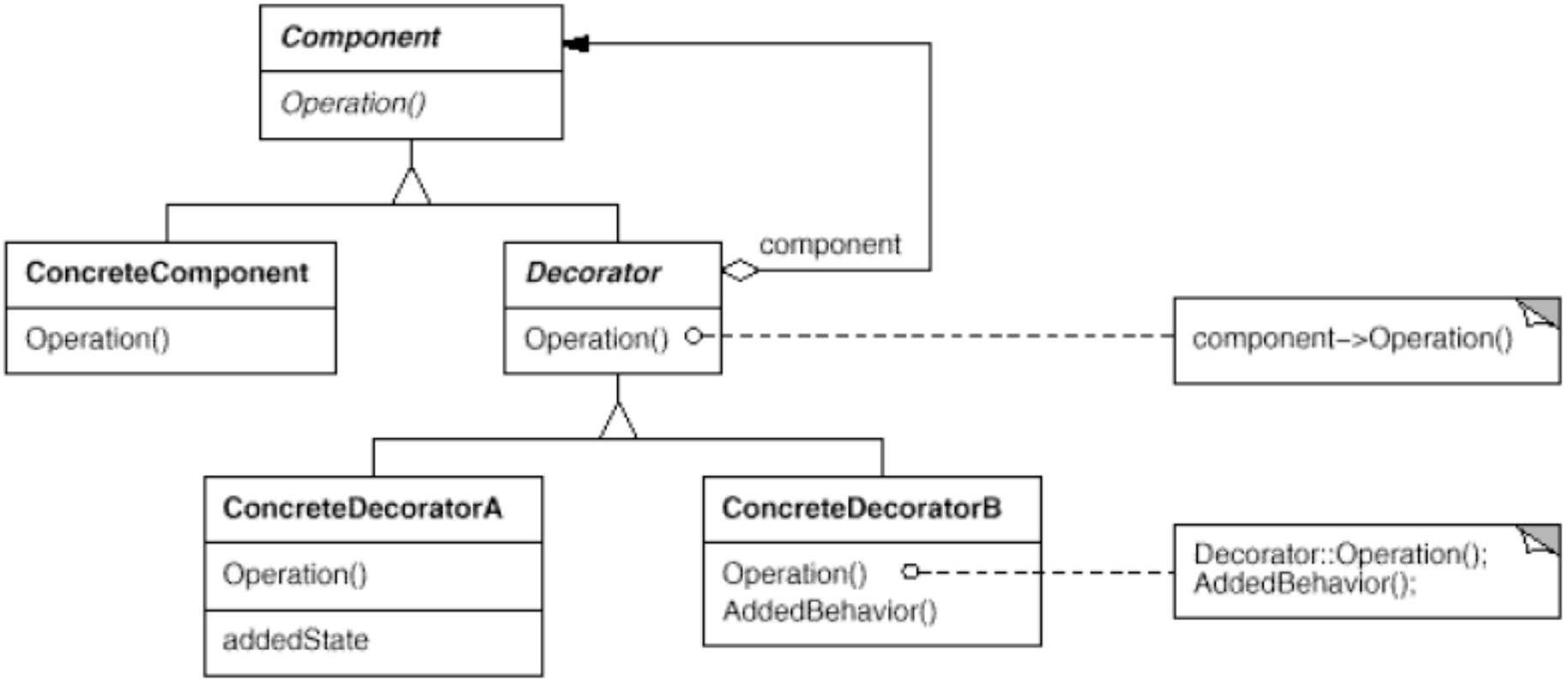
- Clients generally can't tell the difference between a decorated component and an undecorated one, and so
- Clients don't depend at all on the decoration.

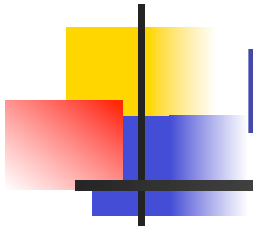


DE Applicability

- 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.

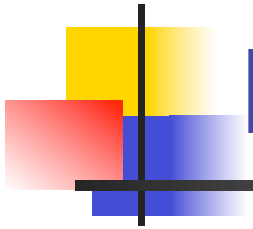
DE Structure





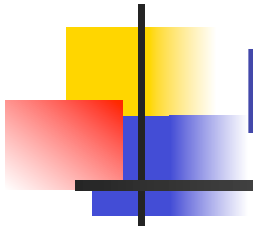
DE Participants

- Component (VisualComponent)
  - defines the interface for objects that can have responsibilities added to them dynamically.
- ConcreteComponent (TextView)
  - defines an object to which additional responsibilities can be attached.
- Decorator
  - maintains a reference to a Component object and defines an interface that conforms to Component's interface.
- ConcreteDecorator (BorderDecorator, …)
  - adds responsibilities to the component.



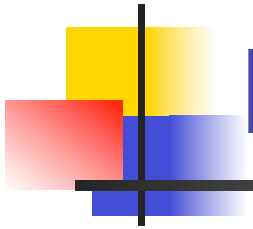
DE Collaborations

- Decorator forwards requests to its Component object, and (optionally) perform additional operations before and after forwarding the request.



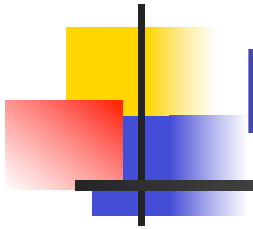
DE Consequences (1)

- Benefits:
  - *More flexibility than static inheritance*
    - responsibilities can be added and removed at run-time
    - inheritance requires creating a new class for each additional responsibility
    - DE also make it easy to add a property twice (e.g. a double border)
    - Inheriting from a Border class twice is error-prone at best.
  - *Avoids feature-laden classes high up in the hierarchy.*
    - offers a pay-as-you-go approach to adding responsibilities.
    - incrementally-nested simple classes vs complex customizable classes



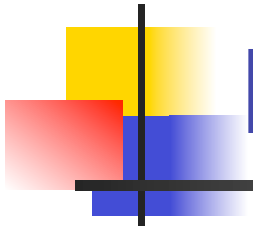
DE Consequences (2)

- DE drawbacks
  - A decorator and its component aren't identical.
    - from an object identity point of view, a decorated component is **not identical** to the component itself.
    - Hence you shouldn't rely on object identity when you use decorators.
  - Lots of little objects.
    - The design often results in systems composed of lots of little objects that all look alike.
    - The objects differ only in the way they are interconnected, not in their class or in the value of their variables.
    - easy to customize but hard to learn and debug!!



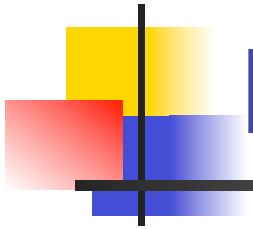
DE Implementation (1)

- Interface conformance.
  - A decorator object's interface must conform to the interface of the component it decorates.
    - ConcreteDecorator classes must therefore inherit from a common class (at least in C++).
- Omitting the abstract Decorator class.
  - when you only need to add one responsibility.
  - when you're dealing with an existing class hierarchy
  - merge Decorator's responsibility for forwarding requests to the component into the ConcreteDecorator.



DE Implementation (2)

- Keeping Component classes lightweight.
  - components and decorators must descend from a common Component class
    - To ensure a conforming interface
    - It's important to keep this common class lightweight
      - it should focus on the interface, not on storing data.
      - data representation should be deferred to subclasses
      - otherwise:
        - decorators too heavyweight to use in quantity.
        - probably concrete subclasses will pay for features they don't need.



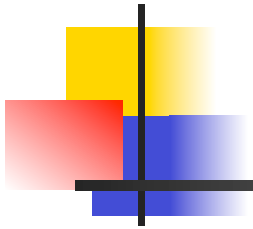
DE Sample Code

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

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

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

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


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

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

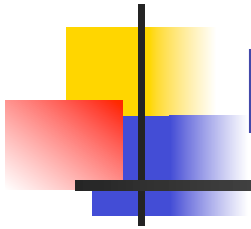
```
class BorderDecorator : public Decorator {
public:
    BorderDecorator(VisualComponent*, int borderWidth);

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

void BorderDecorator::Draw () {
    Decorator::Draw();
    DrawBorder(_width);
}
```

```
window->SetContents(new BorderDecorator(
                        new ScrollDecorator(textView), 1
                    )
                );
```



DE Known Uses and Related Patterns

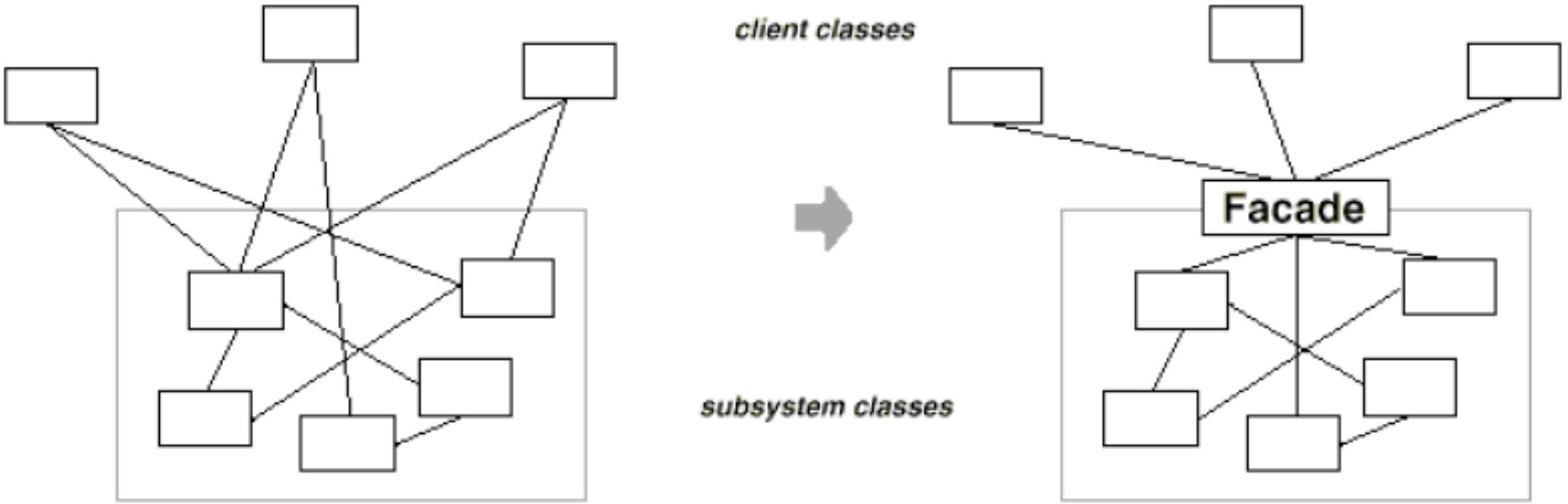
- Known Uses
  - The Java stream zoo… and more
- Related Patterns
  - Adapter
  - Composite
  - Strategy

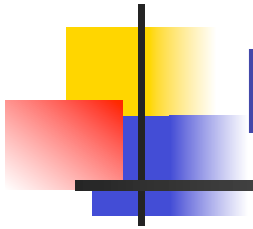


Facade (FA)

- 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.
  - Introduce a **facade** object providing a single, simplified interface to a subsystem.

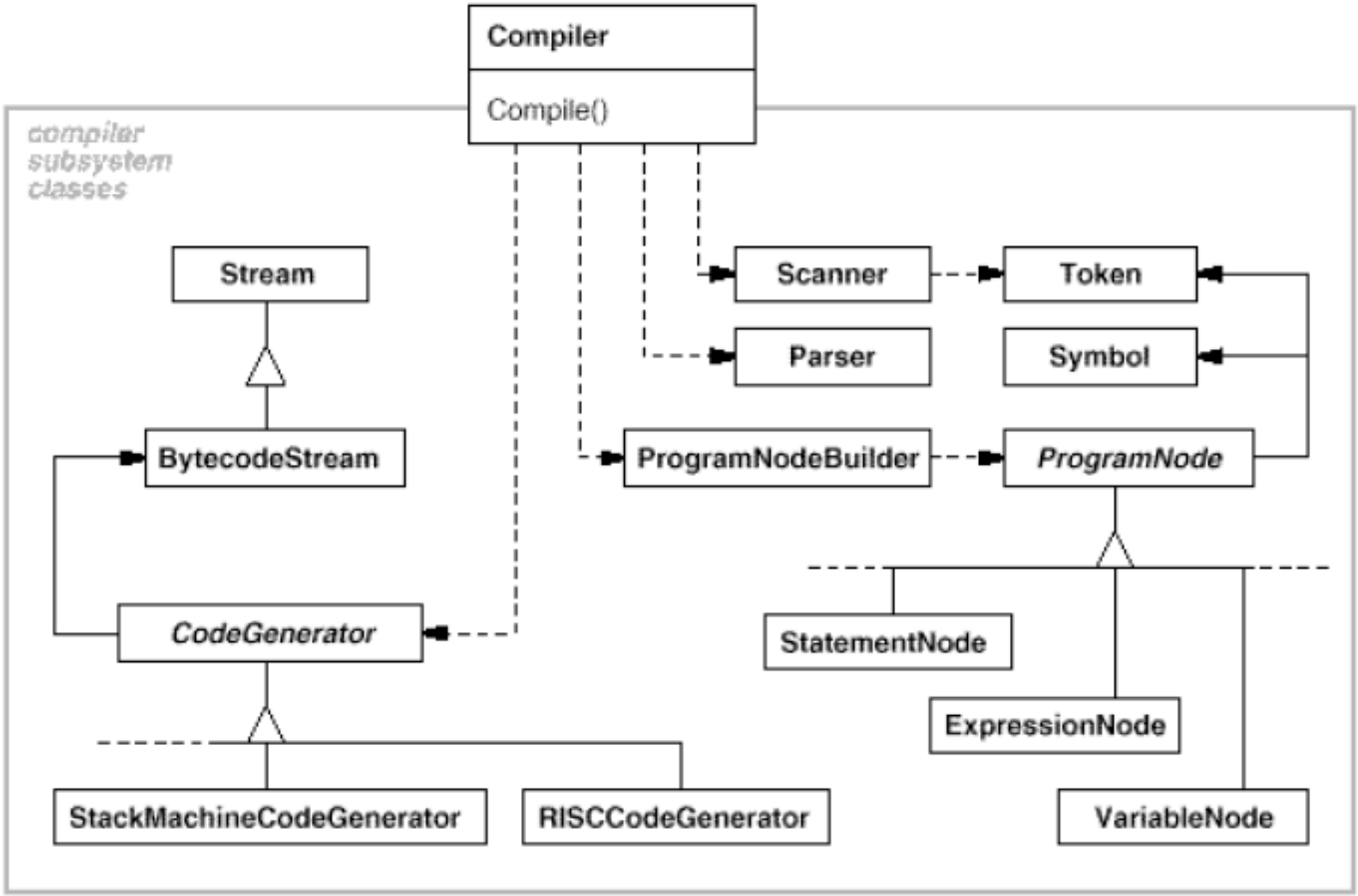
FA Motivation (continued)

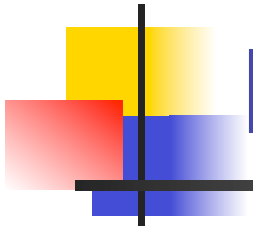




FA Motivation (3)

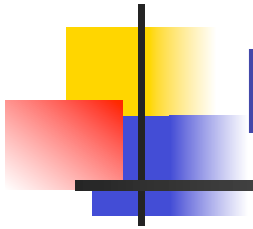
- Consider a develop. tool where applications may access to the compiler subsystem.
  - Scanner, Parser, ProgramNode, BytecodeStream, and ProgramNodeBuilder…
- Most clients of a compiler generally don't care about details
  - they merely want to compile some code.
  - No need for the powerful but low-level interfaces
  - Solution:
    - provide a higher-level unified interface that can shield clients from these classes: a Compiler class.
    - The Compiler class acts as a facade!


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



FA Applicability (1)

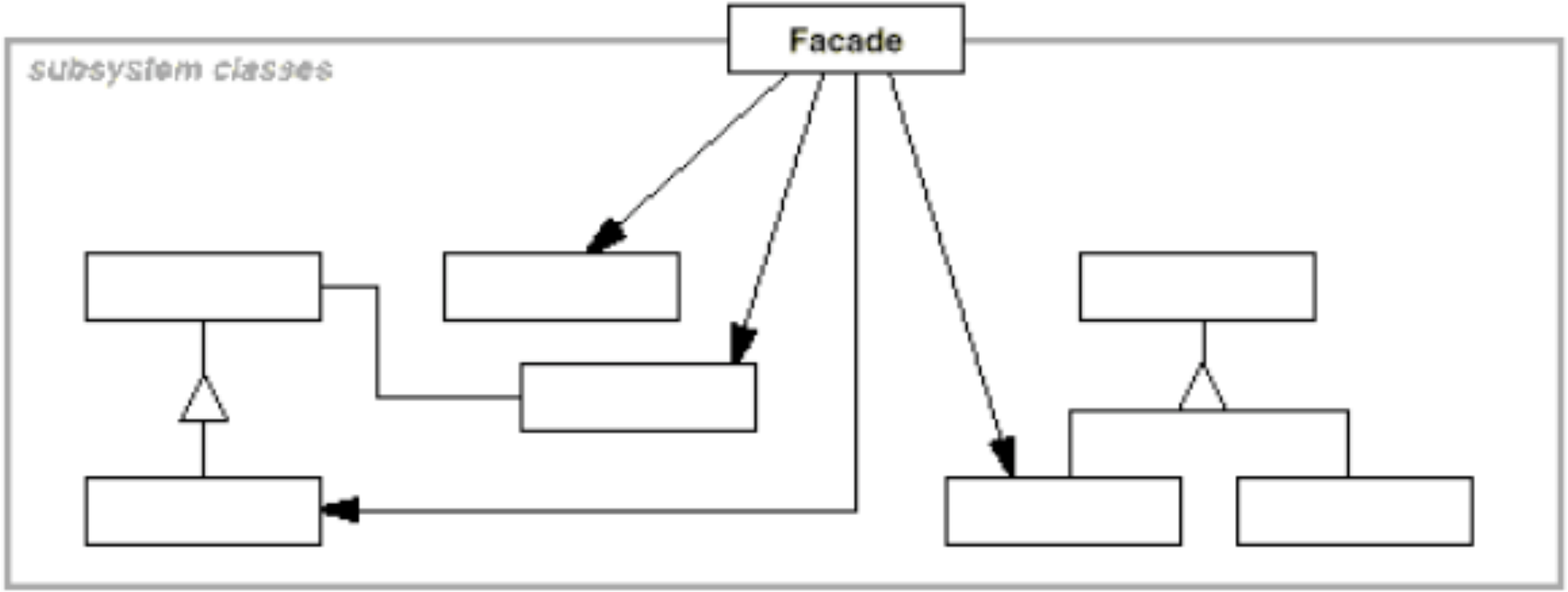
- Use the Facade pattern when:
  - you want to provide a simple interface to a complex subsystem.
    - This makes the subsystem more reusable and easier to customize (the FA hides refactorings)
    - but it also becomes harder to use for clients that don't need to customize
  - You want to provide a simple default view of the subsystem that is good enough for most clients.
    - Only clients needing more customizability will need to look beyond the facade.

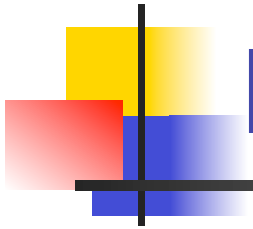


FA Applicability (2)

- Use the Facade pattern when:
  - there are many dependencies between clients and the implementation classes
    - FA decouple the subsystem from clients and other subsystems,
      - => more subsystem independence and portability.
  - you want to layer your subsystems.
    - Use a FA to define an entry point to each subsystem level.
    - Subsystems communicate through FAs
      - => simplification of the dependencies between them

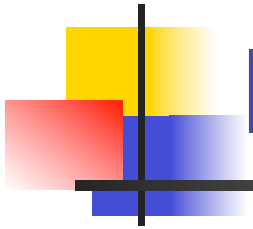
FA Structure





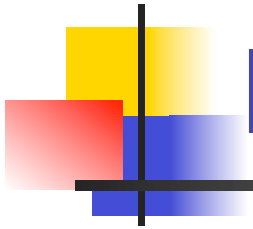
FA Participants

- 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.



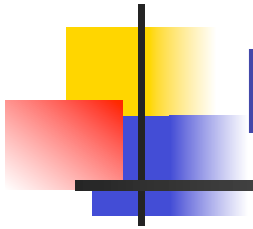
FA Collaborations

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



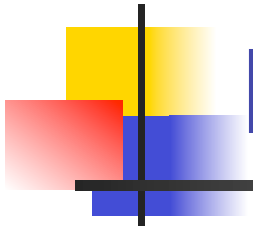
FA Consequences (1)

- shields clients from subsystem components
- reduces the number of objects that clients deal with
  - making the subsystem easier to use.
- promotes weak coupling between the subsystem and its clients.
  - Weak coupling lets you vary the components of the subsystem without affecting its clients.
- help layer a system and the dependencies between objects.



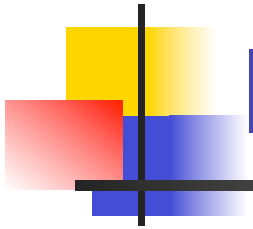
FA Consequences (2)

- can eliminate complex or circular dependencies.
  - Could be important when the client and the subsystem are implemented independently.
- reduces compilation dependencies
  - minimized recompilation when subsystem change.
  - simplification of porting systems to other platforms
- doesn't prevent applications from using subsystem classes if they need to.
  - you can choose between ease of use and generality.



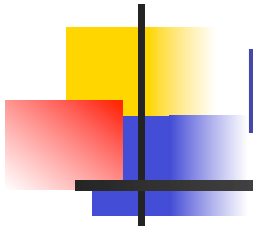
FA Implementation (1)

- Reducing client-subsystem coupling.
  - by making Facade an abstract class
    - Having concrete subclasses for different implementations of a subsystem.
    - Clients interact with the subsystem by the abstract Facade class.
      - clients do not know the subsystem implementation
  - by configuring a FA object with different subsystem objects
    (alternative to subclassing)
    - Customize by simply replace one or more of its subsystem objects.



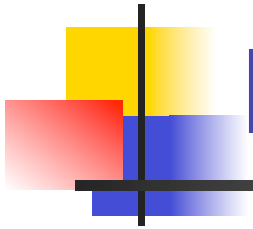
FA Implementation (2)

- Public versus private subsystem classes.
  - Both subsystems and classes
    - have interfaces,
    - encapsulate something
      - a class encapsulates state and operations
      - a subsystem encapsulates classes.
  - The public and private interface of a subsystem:
    - the public interface: classes that all clients can access
    - the private interface: classes for subsystem extenders
  - The FA is part of the public interface, of course, but it's not the only part.
  - There are languages that does not let you hide private subsystem classes



FA Sample Code

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

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

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

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

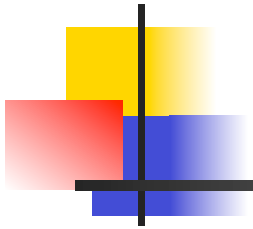


FA Sample Code

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

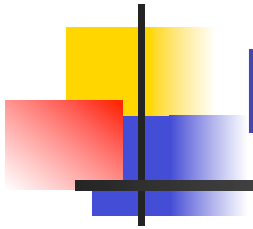
…and more

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

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

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

    parser.Parse(scanner, builder);

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



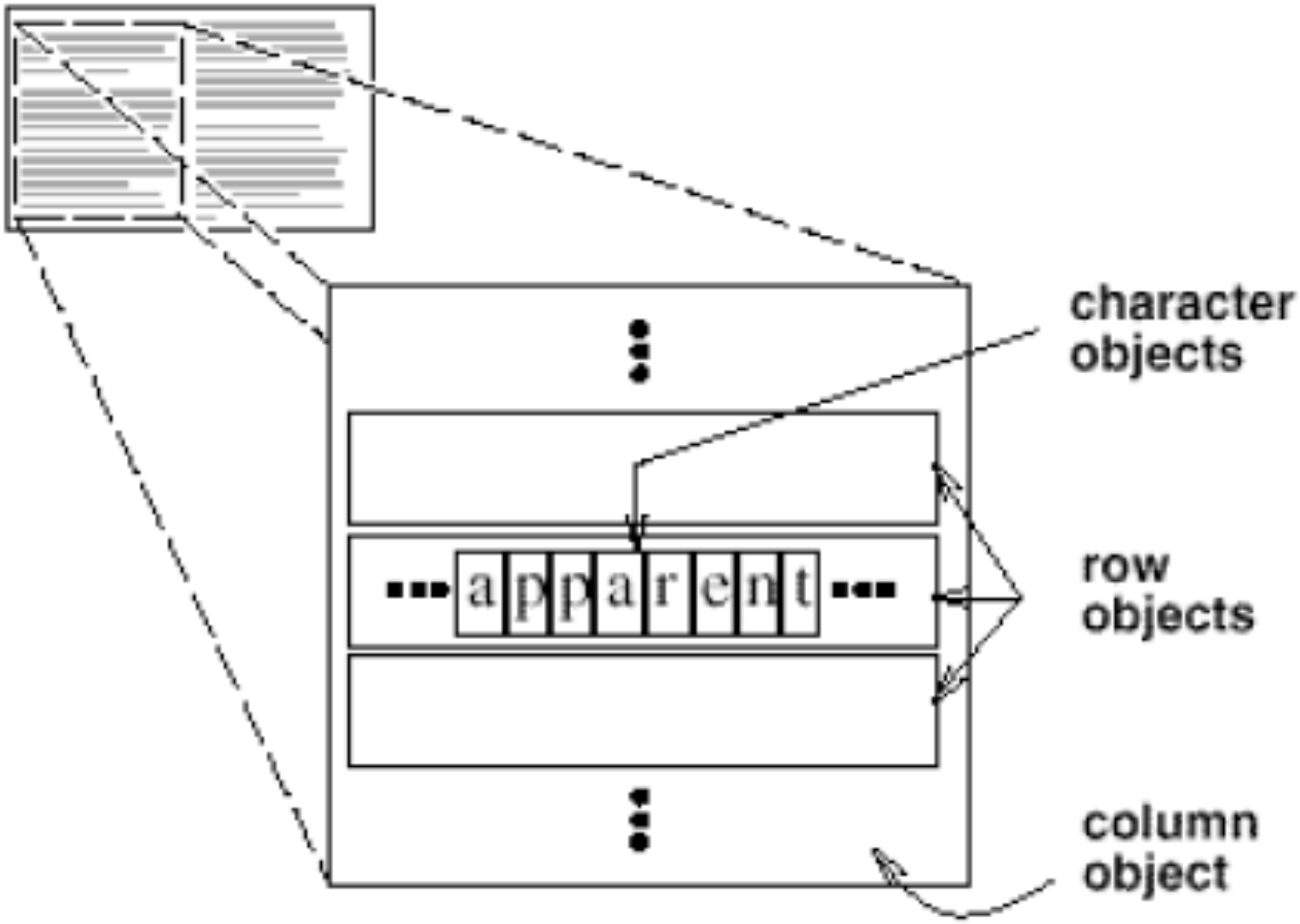
FA Known Uses and Related Patterns

- Known Uses
  - …many applications
- Related Patterns
  - Abstract Factory
  - Mediator
  - Singleton

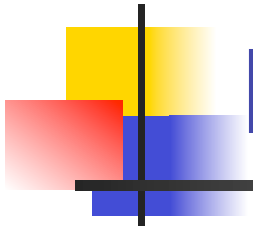


Flyweight (FL)

- Intent
  - Use sharing to support large numbers of fine-grained objects efficiently.
- Motivation
  - Complex object structures can simplify the design of an application but naive implementation can be prohibitively expensive.
  - For example, consider a document editor embedding elements like tables and figures.
    - The application's object structure could mimic the document's physical structure.

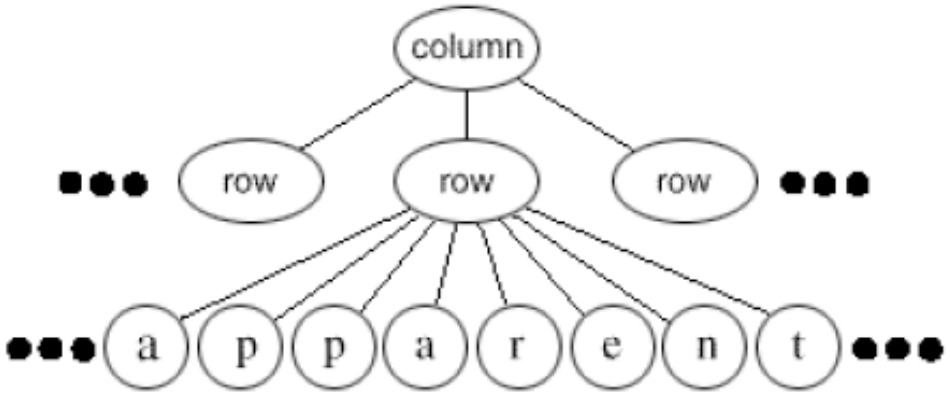


… it is a nice design, but:
… it may easily require hundreds of thousands of character objects,
… which will **consume lots of memory**
… and may incur unacceptable **run-time overhead**!!!!!

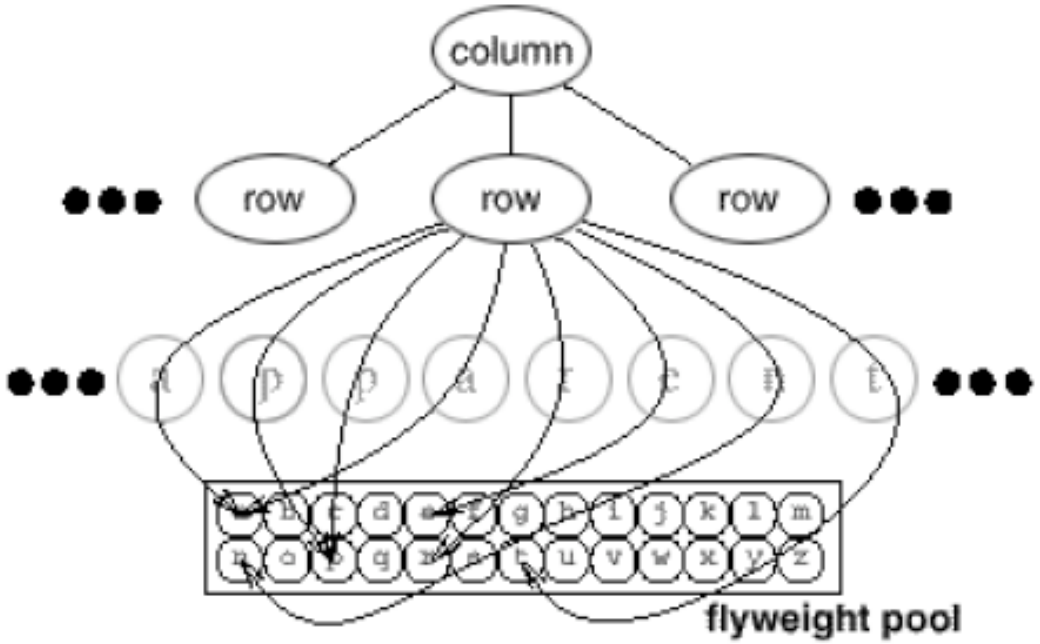


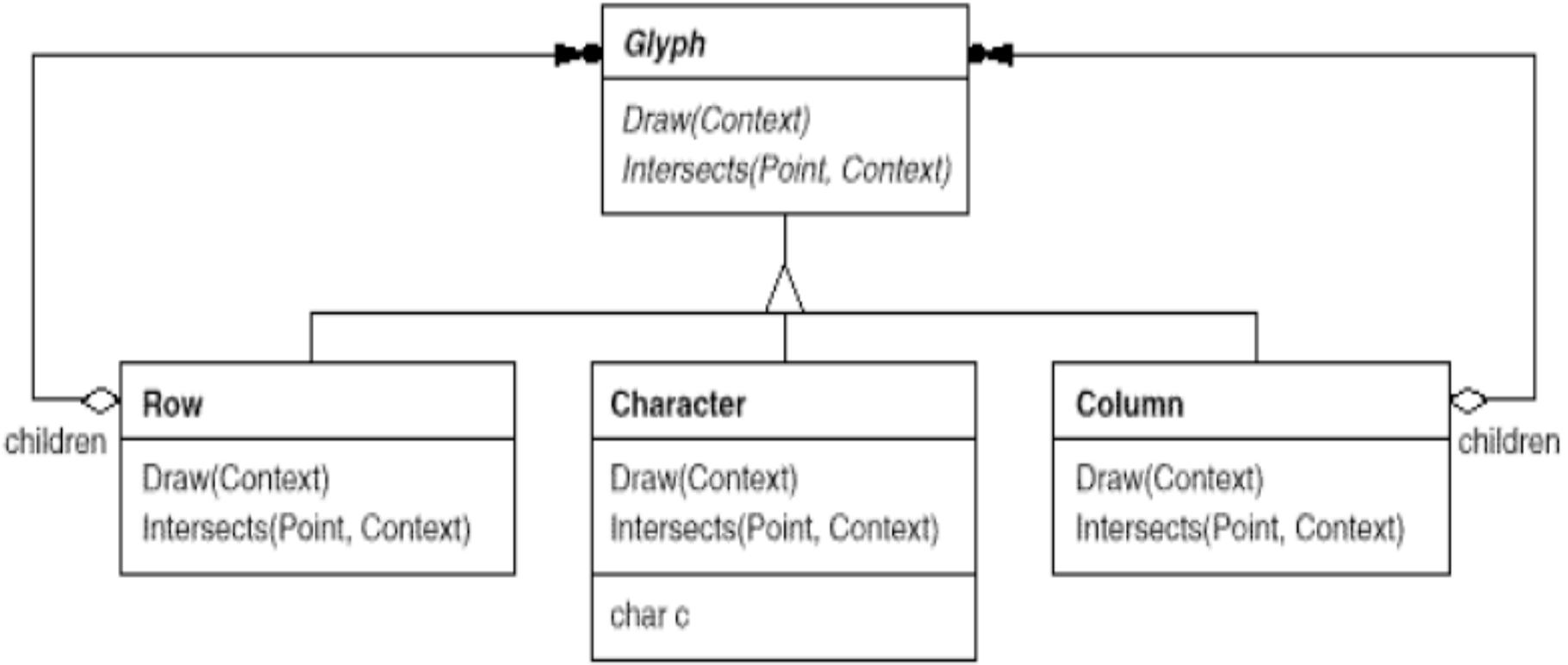
FL Motivation (continued)

- A **flyweight** is a shared object that can be used in multiple contexts simultaneously.
  - independent object in each context
  - FL cannot make assumptions about the context
- **intrinsic** and **extrinsic** state
  - Intrintsic state:
    - stored in the flyweight
    - information that's independent of the flyweight's context
  - Extrinsic state:
    - depends on and varies with the flyweight's context
    - can't be shared.
    - Client pass extrinsic state to the flyweight when it needs it



By using a FL for representing characters becomes:



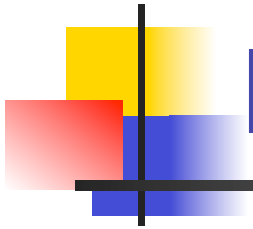


A FL representing the letter "a" only stores the corresponding character code; it doesn't need to store its location or font.

Less different character objects

=>

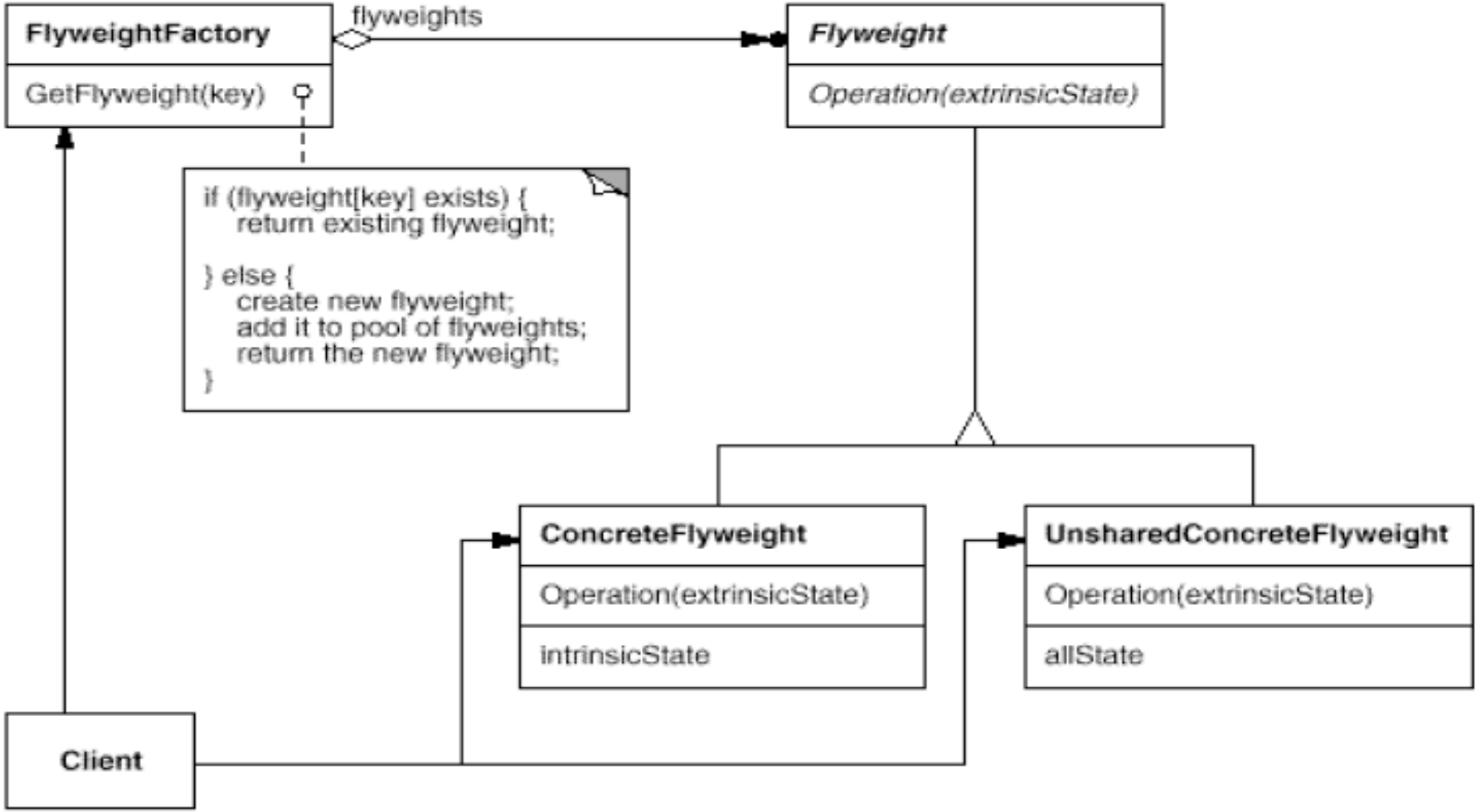
the total number of objects is substantially less



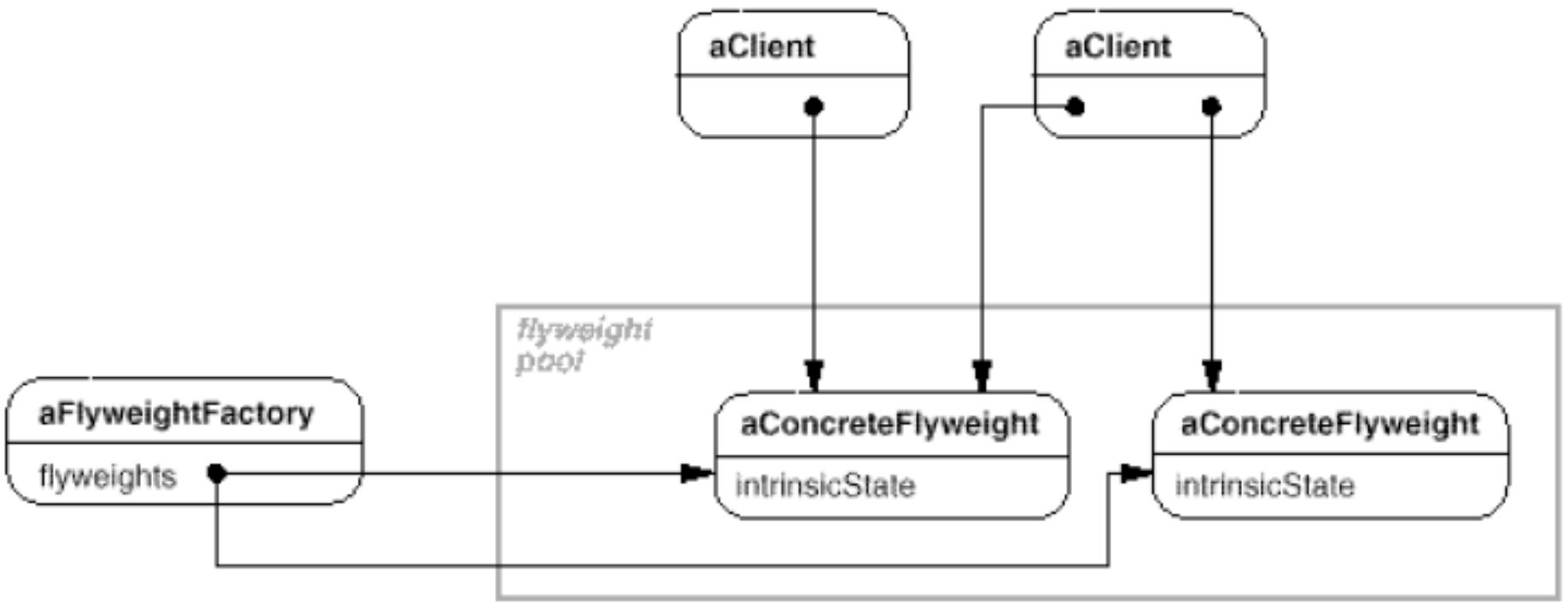
FL Applicability

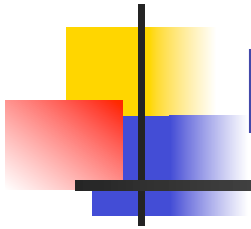
- **all** of the following conditions must hold:
  - An application uses a large number of objects.
  - Storage costs are high because of the sheer quantity of objects.
  - Most object state can be made extrinsic.
  - Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
  - The application doesn't depend on object identity.
    - FL objects may be shared => identity tests will return true for conceptually distinct objects.

FL Structure



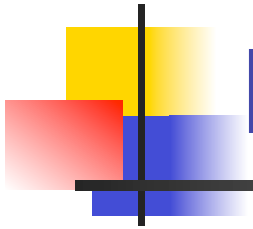
FL Structure





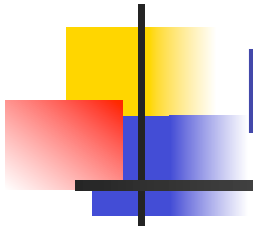
Fl Participants (1)

- 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.
  - must be sharable and independent of the context.
- UnsharedConcreteFlyweight (Row, Column)
  - not all Flyweight subclasses need to be shared.
    - FL enables sharing; it doesn't enforce it!



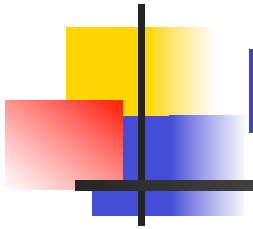
Fl Participants (2)

- FlyweightFactory
  - creates and manages flyweight objects
  - ensures that flyweights are shared properly
    - 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).



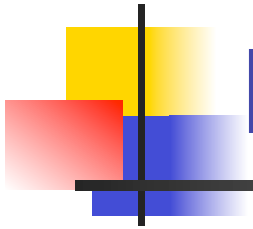
FL Collaborations

- A FL must be characterized as having either intrinsic or extrinsic state.
  - 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



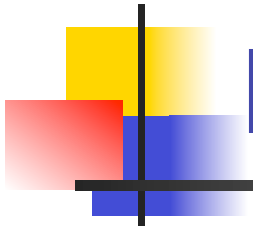
FL Consequences (1)

- FL may introduce run-time costs
  - ***(transferring, finding, and/or computing extrinsic state)***
  - especially for what was formerly stored as intrinsic
  - such costs are offset by space savings
    - Which increase as more flyweights are shared.
- Storage savings 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.



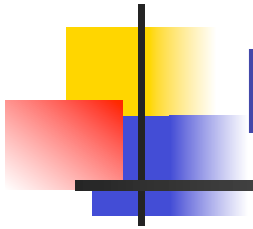
FL Consequences (2)

- The more flyweights are shared, the greater the storage savings.
- The savings increase with the amount of shared state.
  - Then you save on storage in two ways:
    - Sharing reduces the cost of intrinsic state,
    - and you trade extrinsic state for computation time.
- FL pattern is often combined with the Composite pattern (e.g. for representing tree structures)
  - flyweight leaf nodes cannot store a pointer to their parent.
  - the parent pointer is passed to the flyweight as part of its extrinsic state.
  - This has a major impact on how the objects in the hierarchy communicate with each other.



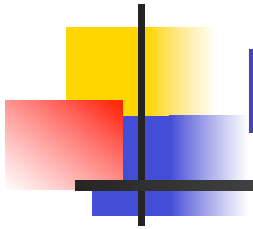
FL Implementation (1)

- Removing extrinsic state.
  - FL applicability is determined by how easy it is to identify extrinsic state and remove it from shared objects.
  - Removing extrinsic state won't help if there are as many different kinds of extrinsic state as there are objects before sharing.
    - Ideally, extrinsic state is computed from a separate object structure having smaller storage requirements.
    - In our document editor, for example, we can store a map of typographic information; storing this information externally to each character object is far more efficient than storing it internally.



FL Implementation (2)

- Managing shared objects.
  - objects are shared => clients shouldn't instantiate them directly.
  - FlyweightFactory lets clients locate a particular flyweight.
    - An associative may ba used to flyweights of interest.
    - e.g. the manager returns the proper flyweight given its code, creating the flyweight if it does not already exist.
  - Sharability could imply some form of reference counting or garbage collection
    - Not necessary if the number of flyweights is fixed and small (e.g., flyweights for the ASCII character set). In that case, the flyweights are worth keeping around permanently.

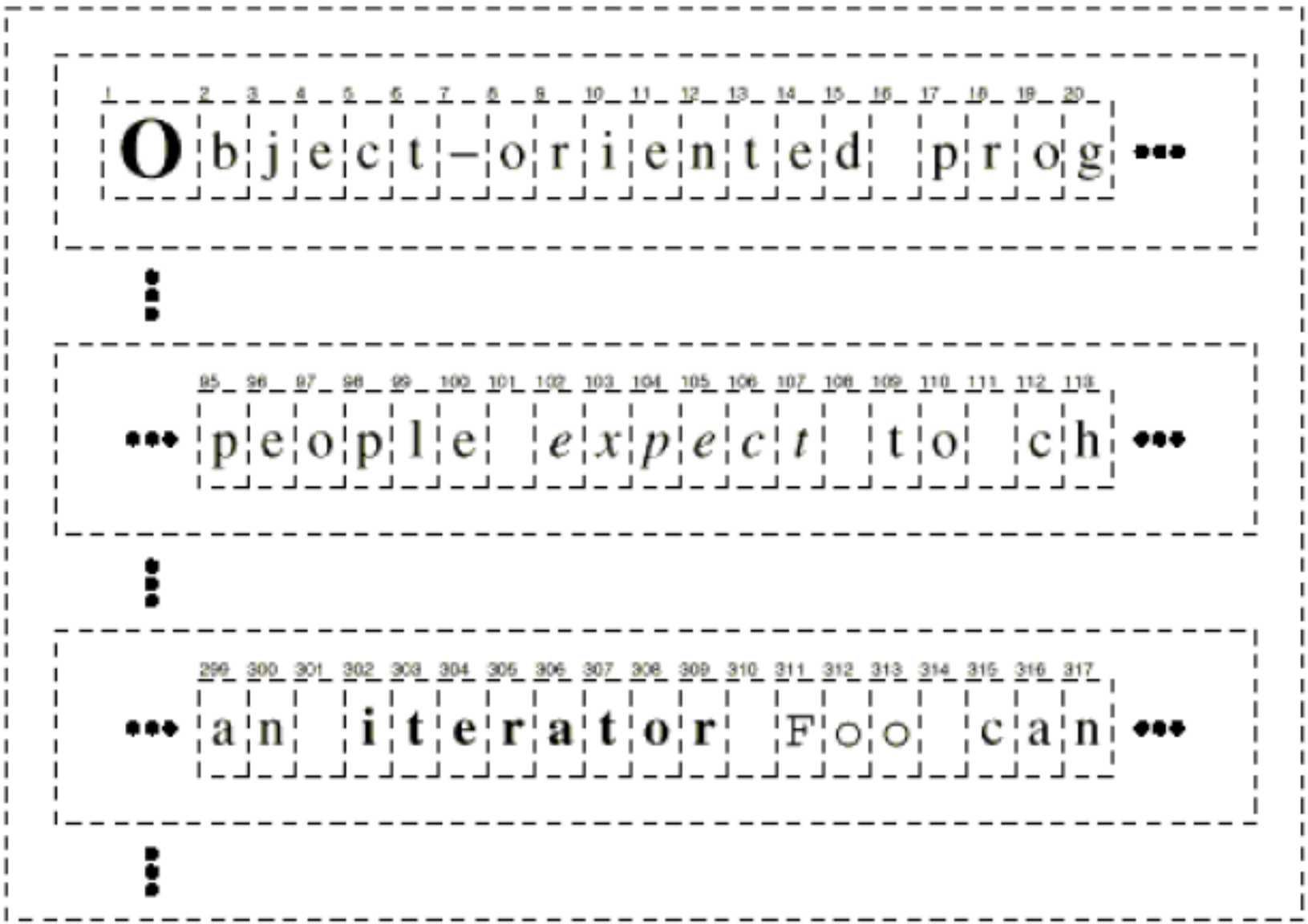


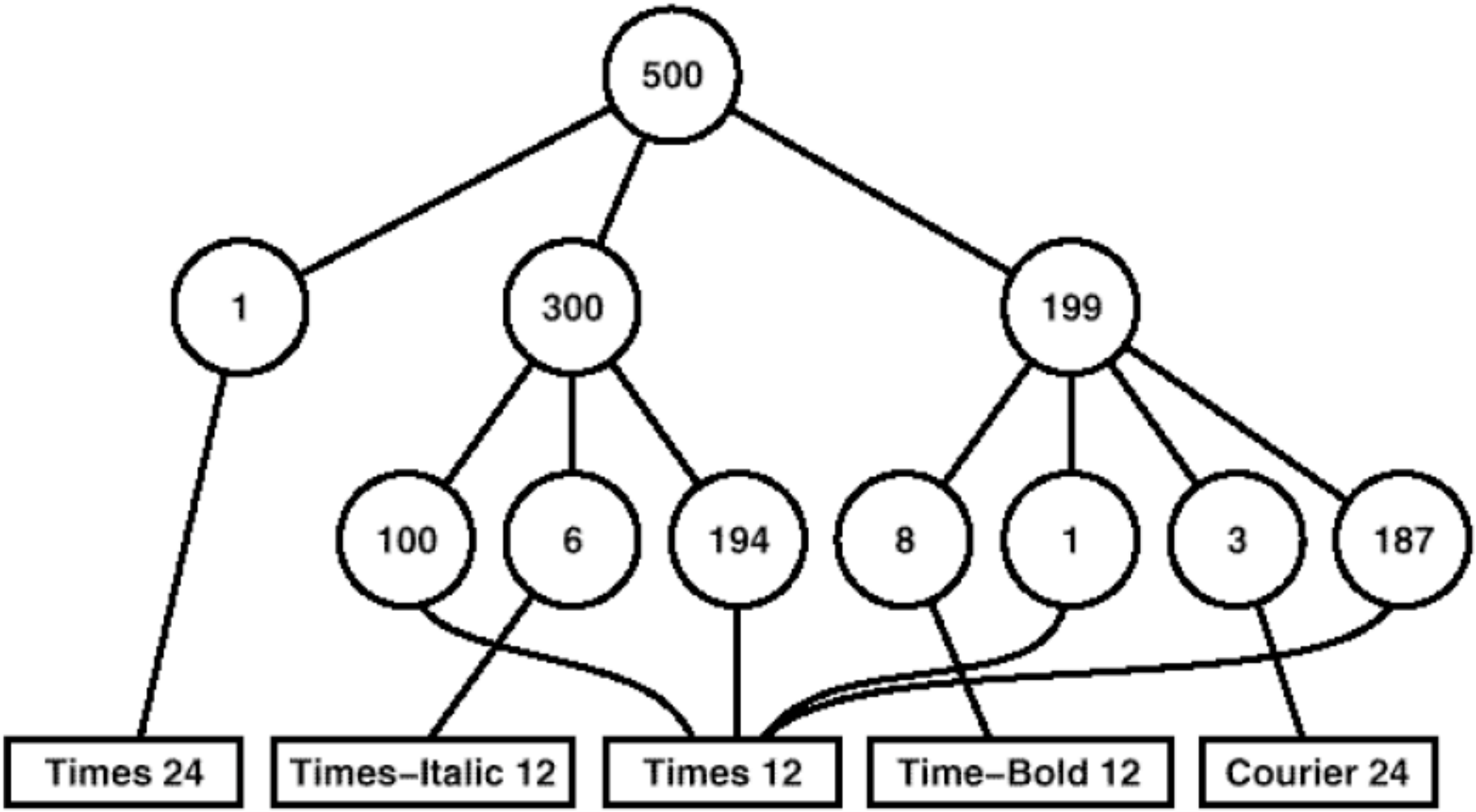
FL Sample Code

- …returning to our document formatter

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

    virtual void Draw(Window*, GlyphContext&);

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

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

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

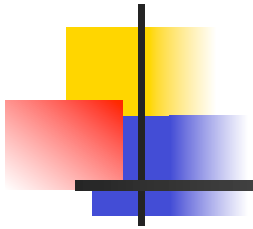
```
class Character : public Glyph {
public:
    Character(char);

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

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

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

    virtual Font* GetFont();
    virtual void SetFont(Font*, int span = 1);
private:
    int _index;
    BTree* _fonts;
};
```


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Object-oriented prog
85 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113
people expect to ch
296 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317
an iterator Foo can


500
1
300
199
100
6
194
8
1
3
187
Times 24
Times–Italic 12
Times 12
Time–Bold 12
Courier 24

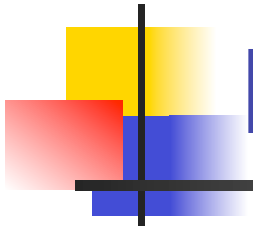
```
const int NCHARCODES = 128;

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

    virtual Character* CreateCharacter(char);
    virtual Row* CreateRow();
    virtual Column* CreateColumn();
    // ...
private:
    Character* _character[NCHARCODES];
};


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

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

    return _character[c];
}
Row* GlyphFactory::CreateRow () {
    return new Row;
}

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



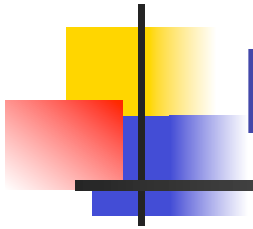
FL Known Uses and Related Patterns

- Known Uses
  - …many applications
- Related Patterns
  - Composite
  - State
  - Strategy



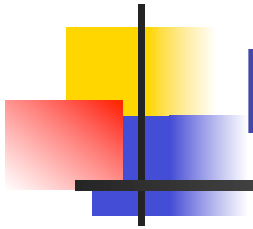
Proxy (PR)

- Intent
  - Provide a surrogate or placeholder for another object to control access to it.
- Also Known As
  - Surrogate



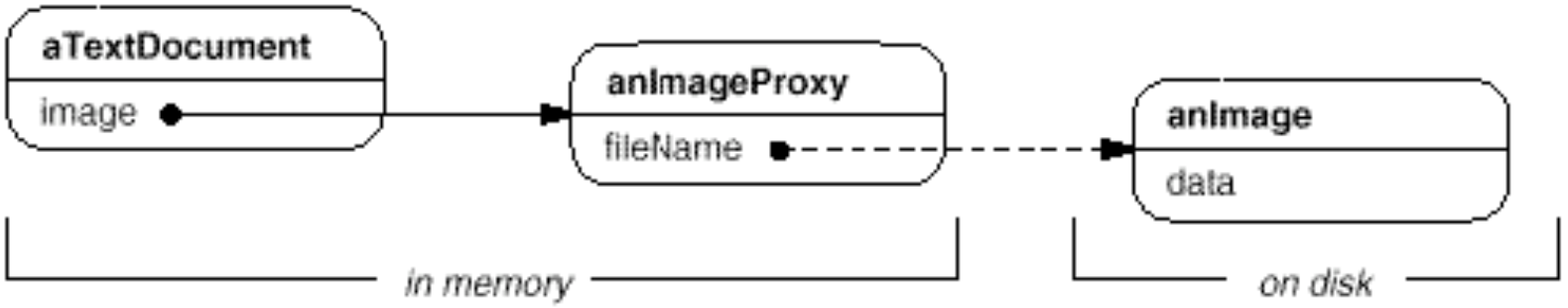
PR Motivation (1)

- 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 (e.g. 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 (not all of these objects will be visible in the document at the same time)

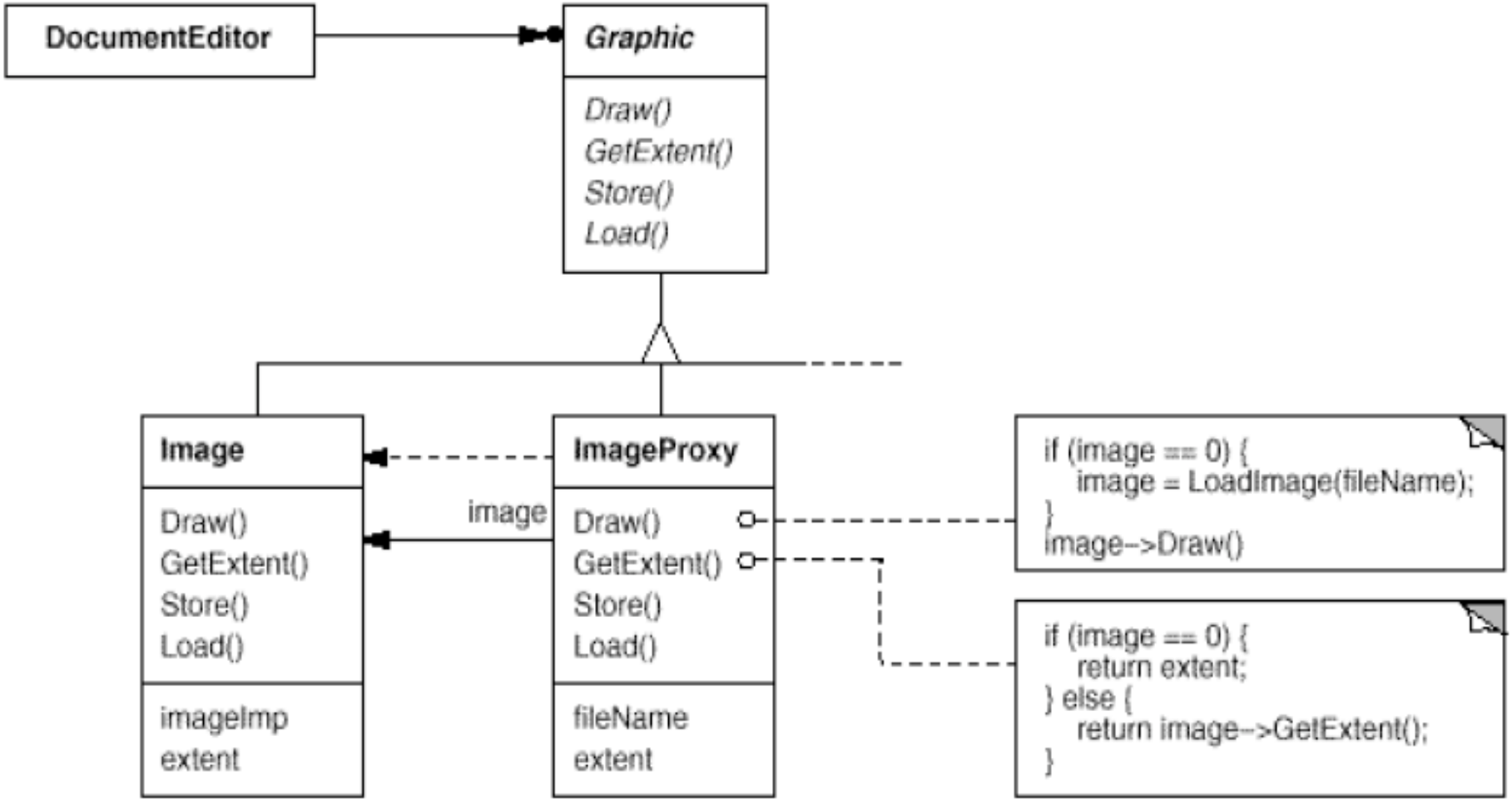


PR Motivation (1)

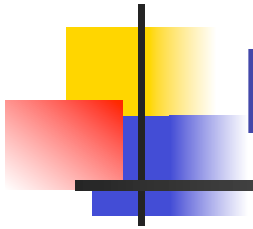
- Should we create each expensive object on demand?
- …or whenever an image becomes visible?
- But what do we put in the document in place of the image?
- And how can we hide this fact? (image is created on demand)
  - use another object, an image proxy, that acts as a stand-in for the real image.
    - The proxy acts just like the image and takes care of instantiating it when it's required.



The image proxy creates the real image only when the document editor asks it to display itself by invoking its Draw operation.

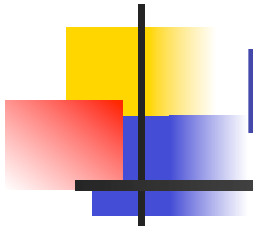


The proxy also stores its **extent**, that is, its width and height.
The extent lets the proxy respond to requests for its size from the formatter without actually instantiating the image.



PR Applicability (1)

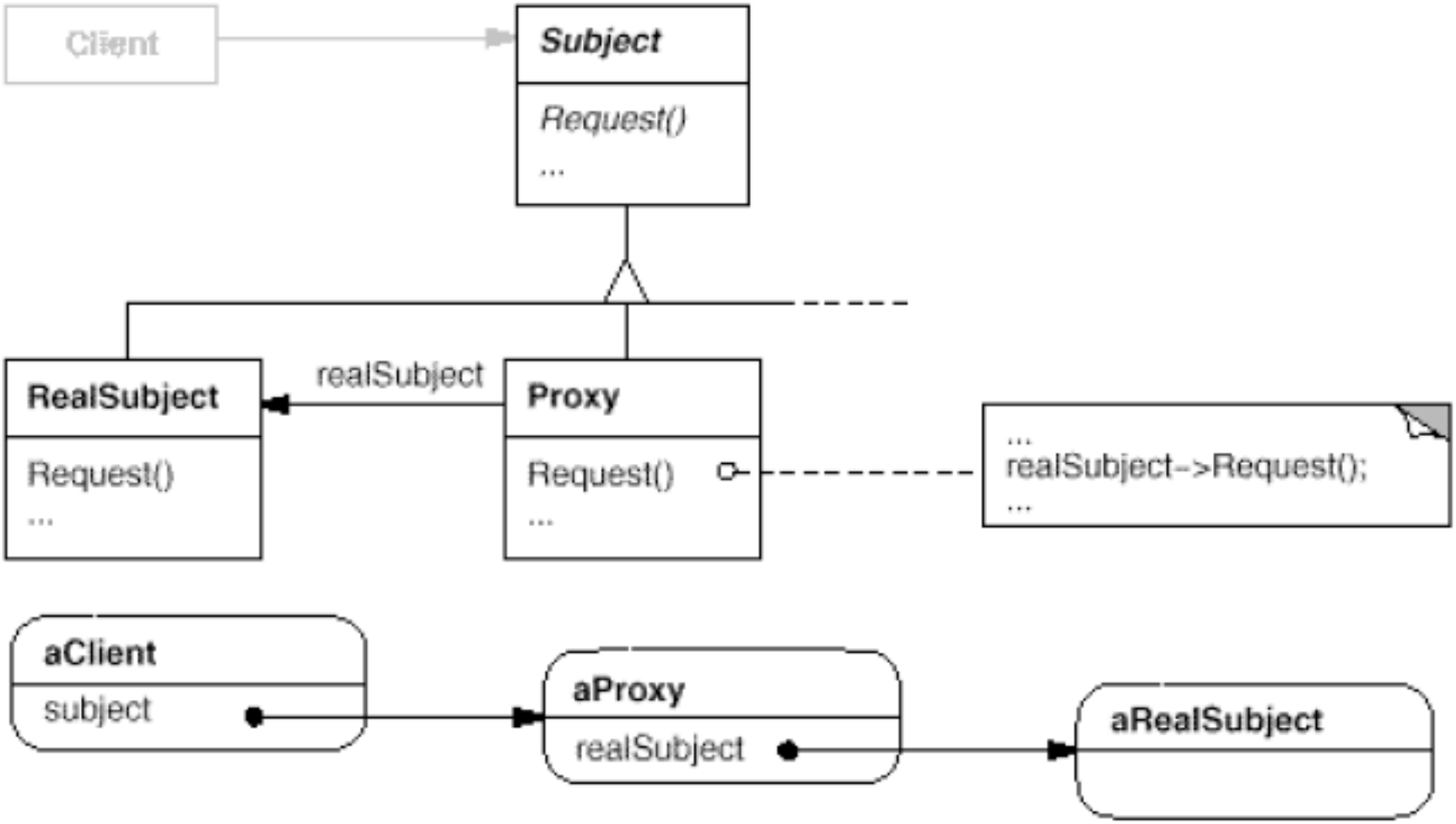
- Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer.
- Common applicability situations:
  - A **remote proxy** provides a local representative for an object in a different address space.
  - A **virtual proxy** creates expensive objects on demand (e.g. ImageProxy)

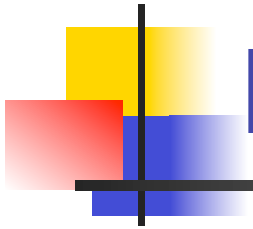


PR Applicability (2)

- A **protection proxy** controls access to the original object
  - (for objects that should have different access rights).
- A **smart reference** is a replacement for a bare pointer that performs additional actions when an object is accessed.
  - to count the number of references to the real object so that it can be freed automatically when there are no more references
  - To load a persistent object into memory when it's first referenced.
  - To check that the real object is locked before it's accessed to ensure that no other object can change it.

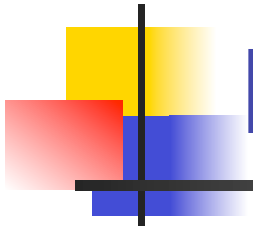
PR Structure





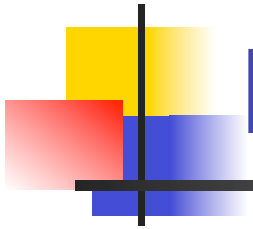
PR Participants (1)

- Proxy (ImageProxy)
  - maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
  - provides an interface identical to Subject's so that a proxy can by substituted for the real subject.
  - controls access to the real subject and may be responsible for creating and deleting it.



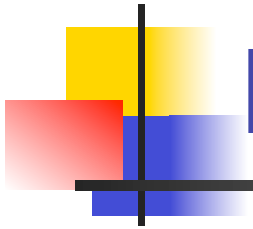
PR Participants (2)

- 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 ImageProxy from the Motivation caches the real image'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.



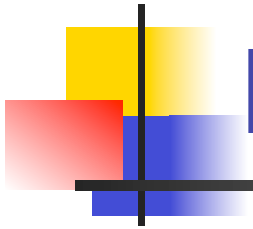
PR Collaborations

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



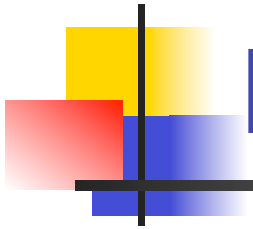
PR Consequences (1)

- The Proxy pattern introduces a level of indirection when accessing an object.
  - 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.



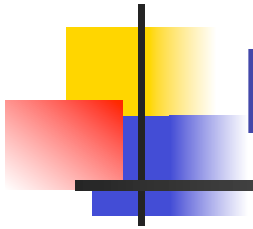
PR Consequences (2)

- Copy-on-write *(creation on demand)*
  - Copying a large and complicated object can be an expensive operation.
    - Copy-on-write can reduce the cost of copying heavyweight subjects significantly.
  - By using a proxy to postpone the copying process,
    - pay the price of copying the object only if it's modified.
  - Requires reference counting.
    - Copying the proxy will do nothing more than increment this reference count.
    - Only when the client requests an operation that modifies the subject does the proxy actually copy it and the subject's reference count is decremented.
    - When the reference count goes to zero, the subject gets deleted.



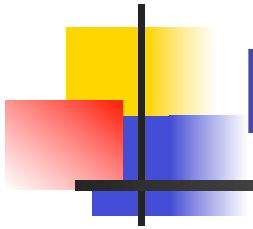
PR Implementation (1)

- Overloading the member access operator in C++.
  - C++ supports overloading operator->, the member access operator.
    - It lets you perform additional work whenever an object is dereferenced
    - This can be helpful for implementing some kinds of proxy
    - the proxy behaves just like a pointer.



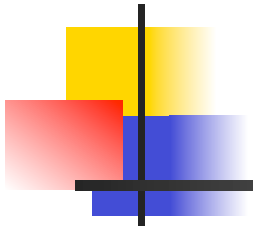
PR Implementation (2)

- Overloading “->” isn't a good solution for every kind of proxy.
  - e.g. if a proxy need to know precisely *which* operation is called
  - Consider the image example
    - the image should be loaded when the Draw operation is called
    - and not whenever the image is referenced.



PR Implementation (3)

- Proxy doesn't always have to know the type of real subject.
  - PR deals with its subject through an abstract interface
    - no need to make a Proxy for each RealSubject
    - the proxy can deal with all RealSubject uniformly.
    - But if Proxies are going to instantiate RealSubjects (e.g. virtual proxy), then they have to know the concrete class.
- Address space-independent object identifiers
  - the way a PR refer to the subject before it's instantiated
    - E.g. a file name for this purpose in the Motivation example.



PR Sample Code

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

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

    virtual const Point& GetExtent() = 0;

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

```
class Image : public Graphic {
public:
    Image(const char* file);  // loads image from a file
    virtual ~Image();
    virtual void Draw(const Point& at);
    virtual void HandleMouse(Event& event);

    virtual const Point& GetExtent();

    virtual void Load(istream& from);
    virtual void Save(ostream& to);
private:
    // ...
};
```

```
class ImageProxy : public Graphic {
public:
    ImageProxy(const char* imageFile);
    virtual ~ImageProxy();

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

    virtual const Point& GetExtent();

    virtual void Load(istream& from);
    virtual void Save(ostream& to);
protected:
    Image* GetImage();
private:
    Image* _image;
    Point _extent;
    char* _fileName;
};
```

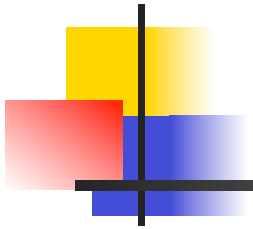
```
ImageProxy::ImageProxy (const char* fileName)  {
    _fileName = strdup(fileName);
    _extent = Point::Zero;  // don't know extent yet
    _image = 0;
}

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

```
const Point& ImageProxy::GetExtent () {
    if (_extent == Point::Zero) {
        _extent = GetImage()->GetExtent();
    }
    return _extent;
}

void ImageProxy::Draw (const Point& at) {
    GetImage()->Draw(at);
}

void ImageProxy::HandleMouse (Event& event) {
    GetImage()->HandleMouse(event);
}
```

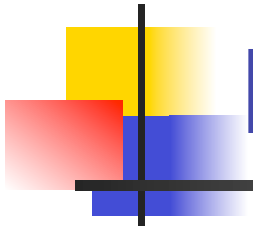
```
void ImageProxy::Save (ostream& to) {
    to << _extent << _fileName;
}

void ImageProxy::Load (istream& from) {
    from >> _extent >> _fileName;
}
```

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

    void Insert(Graphic*);
    // ...
};

TextDocument* text = new TextDocument;
// ...
text->Insert(new ImageProxy("anImageFileName"));
```

PR Known Uses and Related Patterns

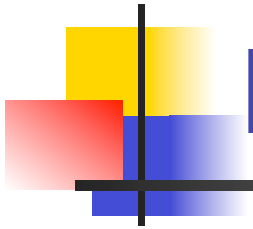


- Known Uses
  - …many applications
- Related Patterns
  - Adapter
  - Decorator



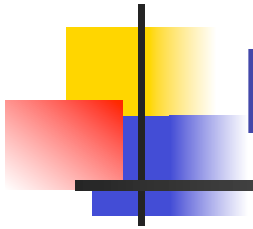
Discussion (1)

- There are a lot of similarities between the structural patterns
  - probably because structural patterns rely on the same small set of language mechanisms for structuring code and objects
    - inheritance for class-based patterns,
    - and object composition for object patterns
- Let's compare and contrast groups of structural patterns



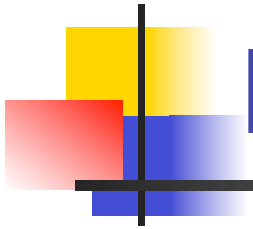
Discussion (2)

- Adapter versus Bridge
  - The key difference is in their intents.
  - Adapter focuses on resolving incompatibilities between two *existing* interfaces.
  - Bridge bridges an abstraction and its (potentially numerous) implementations.
  - AD is necessary when two incompatible classes should work together, to avoid replicating code.
  - BR is necessary when an abstraction must have several implementations, and both may evolve independently.
  - AD makes things work *after* they're designed;
  - BR makes them work *before* they are.



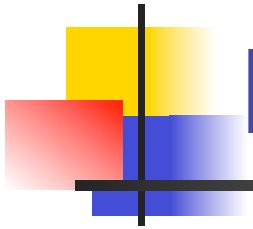
Discussion (3)

- Adapter vs Facade
  - You might think of a FA as an AD to a set of other objects.
    - But FA defines a *new* interface, whereas an AD reuses an old interface.
    - Remember that an AD makes two *existing* interfaces work together as opposed to defining an entirely new one.



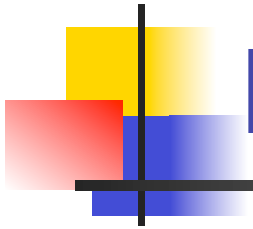
Discussion (4)

- Composite versus Decorator versus Proxy
  - CO and DE are similar
    - both rely on recursive composition
    - Is a DE degenerate CO or vice versa?
      - From the point of view of the DE a CO is a ConcreteComponent.
      - From the point of view of the CO a DE is a Leaf.
    - DE and CO only have different intent!



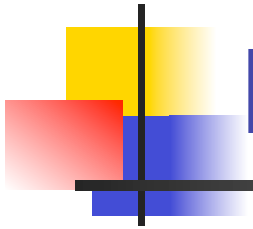
Discussion (5)

- DE is designed to let you add responsibilities to objects without subclassing
- Composite focuses on structuring classes so that many related objects can be treated uniformly, and multiple objects can be treated as one.
  - not on embellishment but representation.
- These intents are distinct but complementary
- CO and DE are often used in concert.



Discussion (6)

- DE and PR are very similar
  - Both describe how to provide a level of indirection to an object,
  - and the implementations keep a reference to another object to which they forward requests.
  - Once again, however, they are intended for different purposes.



Discussion (7)

- Like DE, PR composes an object and provides an identical interface to clients.
- Unlike DE,
  - PR is not concerned with attaching or detaching properties dynamically,
  - PR is not designed for recursive composition.
- PR's intent is to provide a stand-in for a subject when it's inconvenient or undesirable to access the subject directly
- In DE, the component provides only part of the functionality
- In DE recursive composition an essential part while PR focuses on one static relationship (proxy-subject)