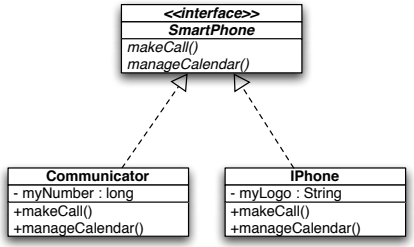
ADDING Dynamically Behavior to Objects

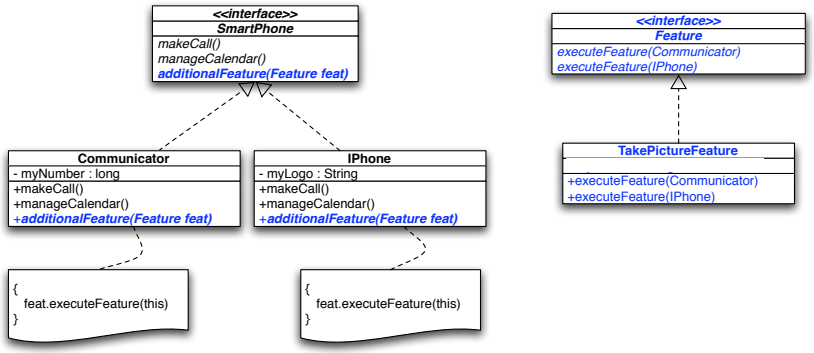
Let's Play with Smart Phones...

Smart Phones. The Challenge... :)



- **clients** may want to add new **features** to these classes, but we are allowed to add **just one method to the hierarchy...**
- **What should we do? :)**

First Solution



Actually what we have is a 2D matrix of features

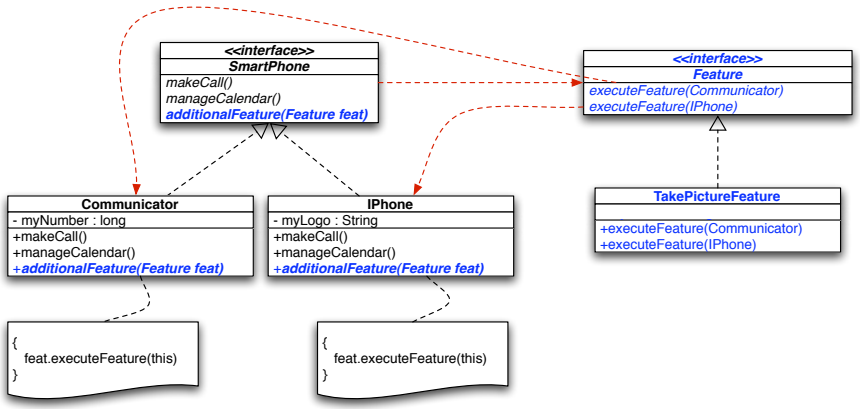
	Features			
Smart Phones		Take Pictures	Video Call	....
	IPhone	X	X	
	Communicator	X	X	
	....			

The Matrix Reveals a Problem...

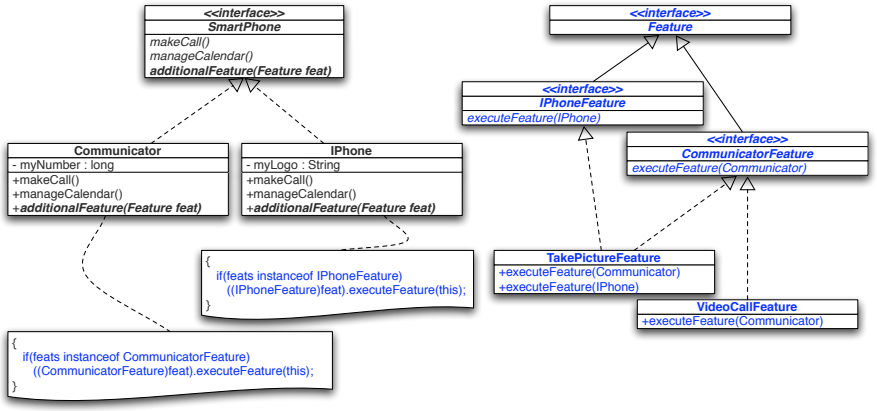
- It is easy add a new **`Feature`**, but hard to add a new **`SmartPhone`**
  - We have to change the entire **`Feature`** hierarchy!!
- ...and even if we change who says that all SmartPhone will have all the additional features?!!
- In other words:

WHAT IF THE MATRIX IS SPARSE?

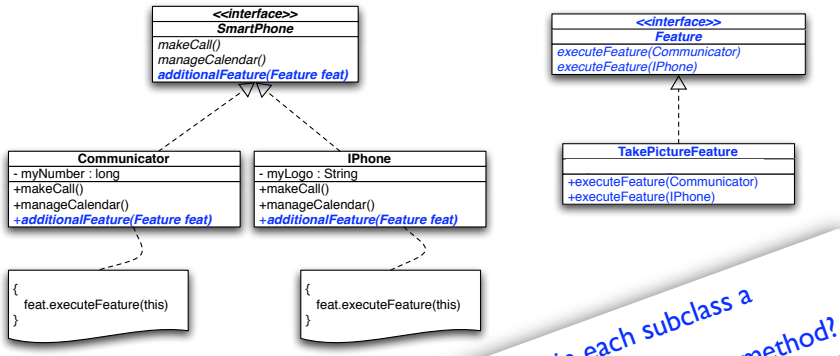
Cyclic Dependencies



Second Solution: Remove Cycles

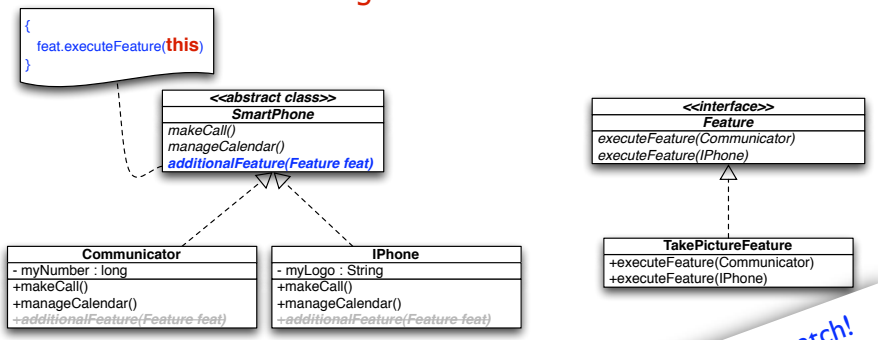


First Solution Revisited



Why have in each subclass a `additionalFeature(Feature)` method?

One Thing That DOES NOT Work



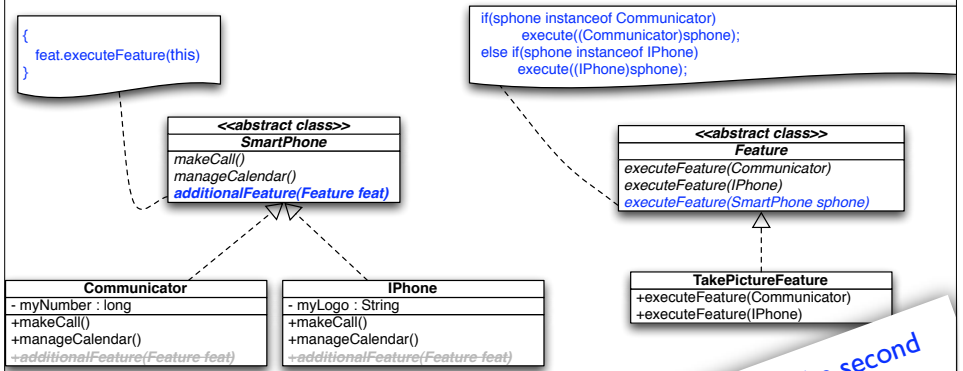
This is the limitation of single dispatch!

(polymorphism works only for caller object)

Factor out `additionalFeature(Feature)` in `SmartPhone`

- transform `SmartPhone` in abstract class (from an interface)
- transform `Feature` in abstract class
- define `executeFeature(SmartPhone)` as a Template Method
  - protected hooks being `executeFeature(IPhone)` and `executeFeature(Communicator)`
- A switch that stays in one place...
  - independently on the number of new features

This Works but...



Switch is needed for the second dispatch

Double Dispatch by Reflection

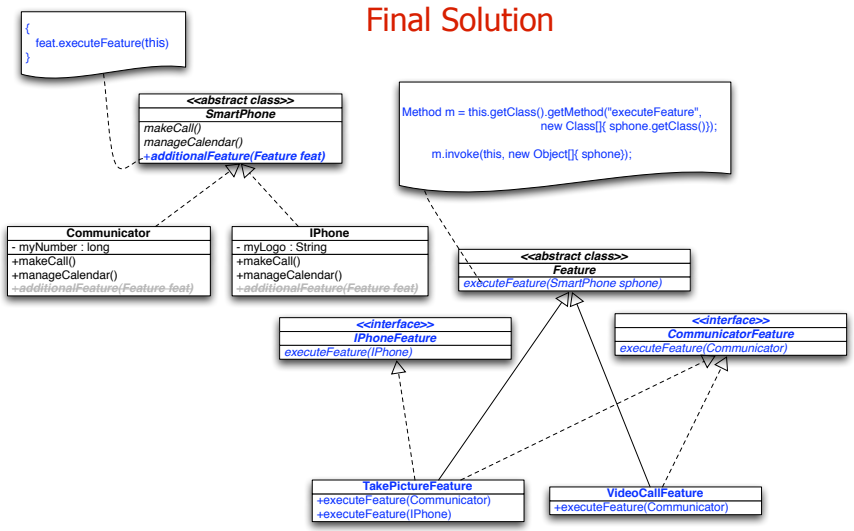
```
abstract class Feature {
  abstract public void executeFeature(IPhone anIPhone);
  abstract public void executeFeature(Communicator aCommunicator);

  public void executeFeature(SmartPhone sphone) throws Exception {
    Method m = this.getClass().
                getMethod("executeFeature",
                          new Class[]{ sphone.getClass()});

    m.invoke(this, new Object[]{ sphone});
  }
}
```

1

2

Get rid of the Switch!

Final Solution



Visitor

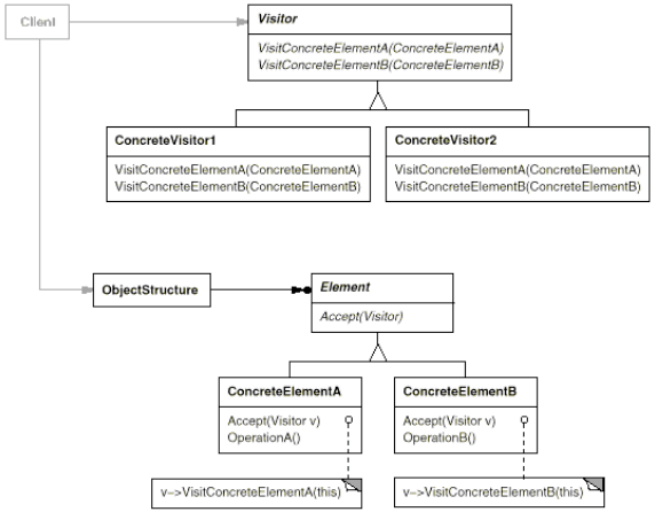
Visitor

- allows new methods to be added to existing hierarchies without modifying the interface of those hierarchies
- Each derivative (i.e. concrete class) of the visited hierarchy has a method in the Visitor hierarchy
- Used for double dispatch:
  - i.e. a double polymorphic dispatch
- **Typical Usage**: generate various **reports** by walking through large data structures

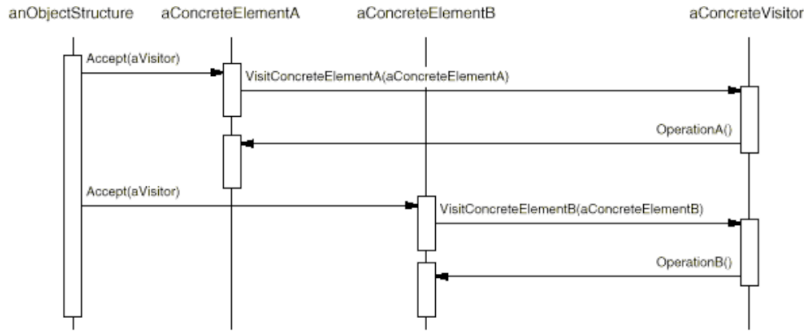
You want to use it when...

- Many distinct and unrelated operations need to be performed on objects in an object structure and you don't want to "pollute" their classes with these operations.
- The classes defining the object structure rarely change, but you often want to define new operations over the structure

Structure



Collaborations



Double Dispatch

- It means that operations get executed depending on the kind of request and types of two receivers, NOT one.
- some programming languages support this directly
  - e.g. Lisp
- Not all programming languages support it directly
  - like Java, C#, C++

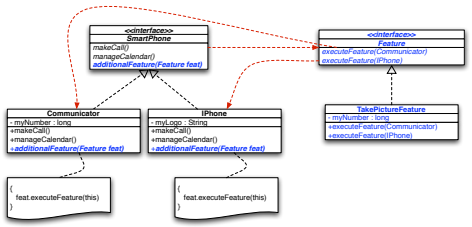
Object Traversal

- Responsibility can fall on:
  1. the structure
  2. the visitor
  3. a separate iterator
- Most common is to use the structure itself, but an iterator is used just as effectively.
- The visitor is used least often to do it, because traversal code often gets duplicated.

Consequences

- Adding new operations is easy!
- Gathers related operations and separates unrelated ones
  - hmmm.... this is not necessarily a positive aspect!
  - simplifying classes defining elements and algorithms defined by visitors.
- Adding new `ConcreteElement` classes is hard.
- Forces you to provide public operations that access an element's internal state, which may compromise encapsulation

Issue of Cyclic Dependencies

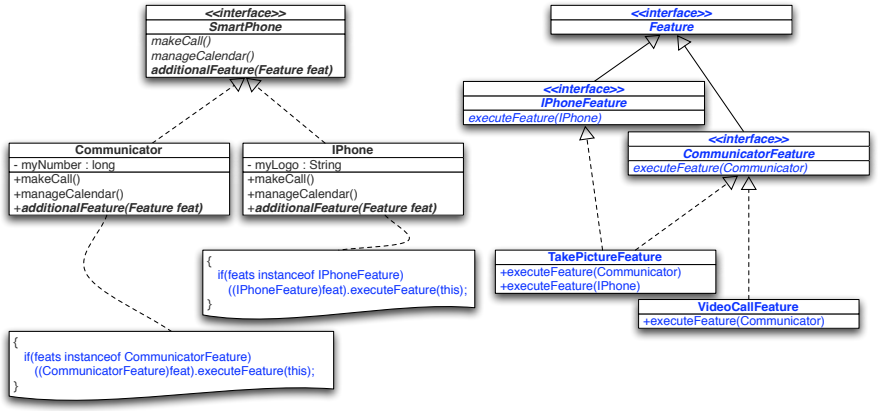


- Bidirectional Dependency
  - Visited hierarchy depends on the base class of the visitor hierarchy
  - base class of the visitor hierarchy depends on each derivative of the visited hierarchy
- **Cycle of dependencies ties all visited derivatives together**
  - difficult to compile incrementally
  - difficult to add new derivatives of the visited hierarchy

Acyclic Visitor

- used for a volatile hierarchy
  - new derivatives
  - quick compilation time is needed
- **Acyclic Visitor** breaks the dependency cycle by making the visitor base class degenerate
  - i.e. with no methods
- **Acyclic Visitor** is like a **sparse matrix**!

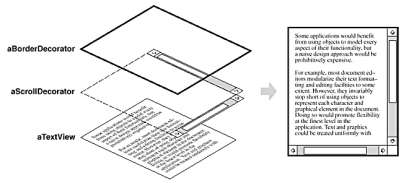
Acyclic Visitor on Example



A Class Inflation Problem...

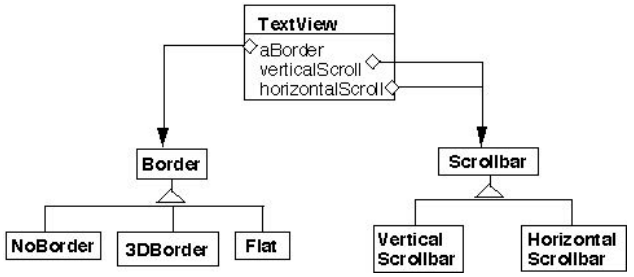
Motivation

- A TextView has 2 features:
  - borders:
    - 3 options: none, flat, 3D
  - scroll-bars:
    - 4 options: none, side, bottom, bo



- How many Classes?
  - 3 x 4 = **12** !!!
    - e.g. TextView, TextViewWithNoBorder&SideScrollbar, TextViewWithNoBorder&BottomScrollbar, TextViewWithNoBorder&Bottom&SideScrollbar, TextViewWith3DBorder, TextViewWith3DBorder&SideScrollbar, TextViewWith3DBorder&BottomScrollbar, TextViewWith3DBorder&Bottom&SideScrollbar, ...

Solution 1: Use Object Composition

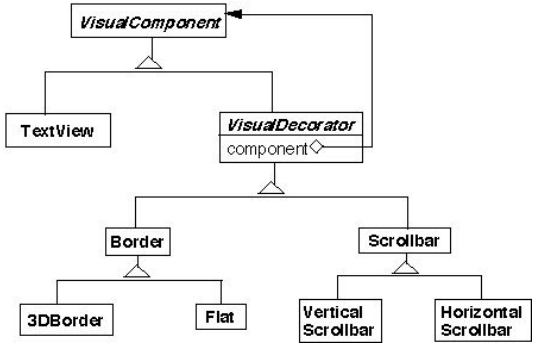


Solution 1: The Source-Code

```
class TextView {
   Border myBorder;
   ScrollBar verticalBar;
   ScrollBar horizontalBar;

   public  void draw() {
      myBorder.draw();
      verticalBar.draw();
      horizontalBar.draw();
      //  code to draw self . . .
   }
   // etc.
}
```

Is it Open-Closed?

Solution 2: Change the Skin, not the Guts!



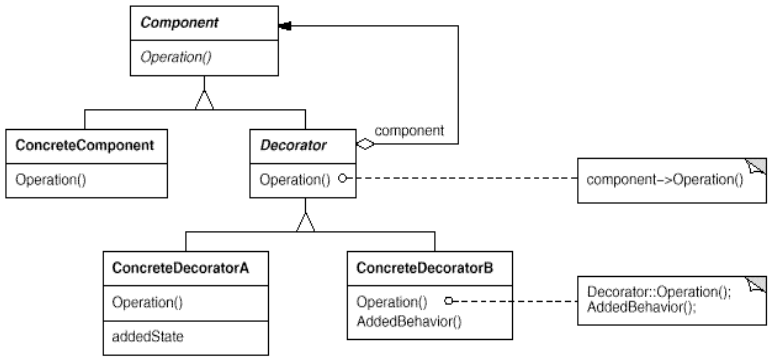
- TextView has **no** borders or scrollbars!
- Add borders and scrollbars on top of a TextView

Decorator Pattern

Basic Aspects

- Intent
  - Add responsibilities to a particular object rather than its class
    - Attach additional responsibilities to an object dynamically.
  - Provide a flexible alternative to subclassing
- Also Known As
  - Wrapper
- Applicability
  - Add responsibilities to objects transparently and dynamically
    - i.e. without affecting other objects
  - Extension by subclassing is impractical
    - may lead to too many subclasses

Structure



Participants & Collaborations

- Component
  - defines the interface for objects that can have responsibilities added dynamically
- ConcreteComponent
  - the "bases" object to which additional responsibilities can be added
- Decorator
  - defines an interface conformant to Component's interface
    - for transparency
  - maintains a reference to a Component object
- ConcreteDecorator
  - adds responsibilities to the component

Consequences

- More flexibility than static inheritance
  - allows to mix and match responsibilities
  - allows to apply a property twice
- Avoid feature-laden classes high-up in the hierarchy
  - "pay-as-you-go" approach
  - easy to define new types of decorations
- Lots of little objects
  - easy to customize, but hard to learn and debug
- A decorator and its component aren't identical
  - checking object identification can cause problems
    - e.g. `if ( aComponent instanceof TextView ) blah`

Implementation Issues

- Keep Decorators lightweight
  - Don't put data members in `VisualComponent`
  - use it for shaping the interface
- Omitting the abstract Decorator class
  - if only one decoration is needed
  - subclasses may pay for what they don't need