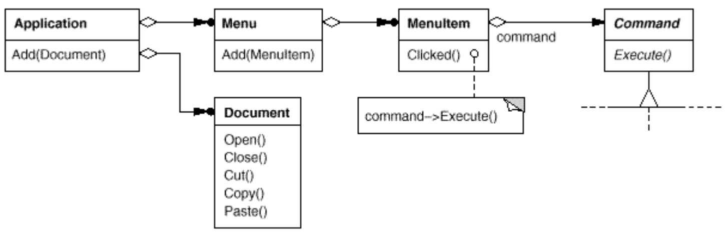
Well-Mannered Dealing of Requests

Command Pattern

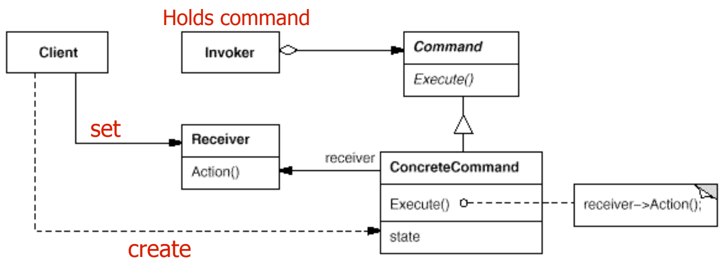
Menu Items Use Commands



Basic Aspects

- Intent
  - Encapsulate requests as objects, letting you to:
    - parameterize clients with different requests
    - queue or log requests
    - support undoable operations
- Applicability
  - Parameterize objects
    - replacement for callbacks
  - Specify, queue, and execute requests at different times
  - Support undo
    - recover from crashes → needs undo operations in interface
  - Support for logging changes
    - recover from crashes → needs load/store operations in interface
  - Model transactions
    - structure systems around high-level operations built on primitive ones
    - common interface ⇒ invoke all transaction same way

Structure

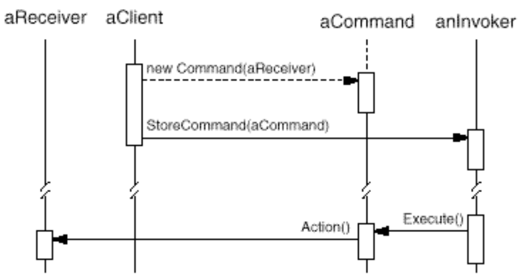


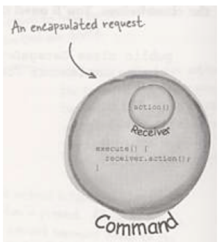
Transforms: concreteReceiver.**action()** in command.**execute()**

Participants

- Command
  - declares the interface for executing the operation
- ConcreteCommand
  - binds a request with a concrete action
- Invoker
  - asks the command to carry out the request
- Receiver
  - knows how to perform the operations associated with carrying out a request.
- Client
  - creates a ConcreteCommand and sets its receiver

Collaborations



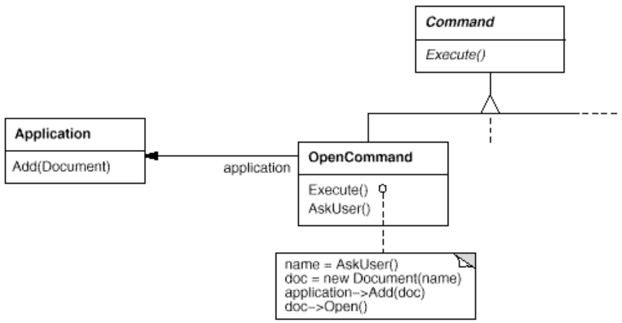


- Client → ConcreteCommand
  - creates and specifies receiver
- Invoker → ConcreteCommand
- ConcreteCommand → Receiver

Consequences

- Decouples Invoker from Receiver
- Commands are **first-class objects**
  - can be manipulated and **extended**
- Composite Commands
  - see also Composite pattern
- Easy to add new commands
  - Invoker does not change
  - it is Open-Closed
- Potential for an excessive number of command classes

Example: Menu Callbacks



Intelligence of Command objects

- "Dumb"
  - delegate everything to Receiver
  - used just to decouple Sender from Receiver
- "Genius"
  - does everything itself without delegating at all
  - useful if no receiver exists
  - let ConcreteCommand be independent of further classes
- "Smart"
  - find receiver dynamically

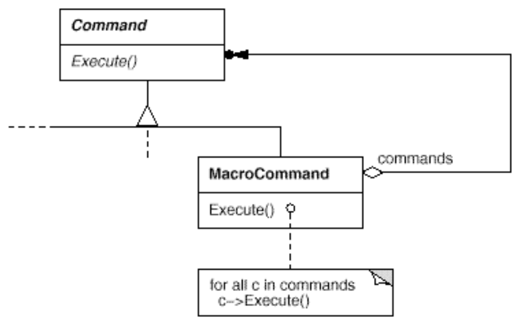
Undoable Commands

- Need to store additional state to reverse execution
  - receiver object
  - parameters of the operation performed on receiver
  - original values in receiver that may change due to request
    - receiver must provide operations that makes possible for command object to return it to its prior state
- History list
  - sequence of commands that have been executed
    - used as LIFO with reverse-execution ⇒ undo
    - used as FIFO with execution ⇒ redo
  - Commands may need to be copied
    - when state of commands change by execution

C++: Commands and Templates

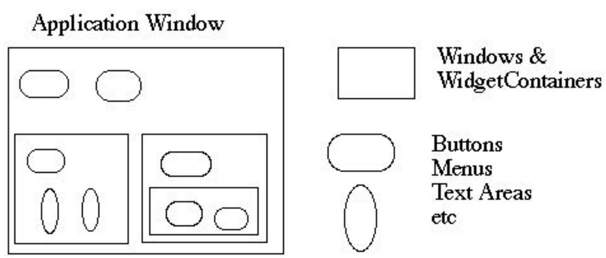
- Avoids subclassing for every kind of action and receiver
- Usable for simple commands
  - don't require arguments in receiver
  - are not undoable
- See example
- Works only for simple commands!
  - if action needs parameters or command must keep state use a Command subclass!

Composed Commands



Composite Pattern

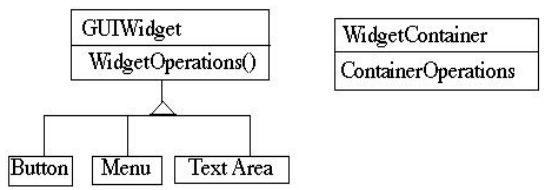
Motivation



- GUI Windows and GUI elements
  - How does the window hold and deal with the different items it has to manage?
  - Widgets are different that WidgetContainers

Implementation Ideas

- Nightmare Implementation
  - for each operation deal with each category of objects individually
  - no uniformity and no hiding of complexity
  - a lot of code duplication
- Program to an Interface
  - uniform dealing with widget operations
  - but still containers are treated different



Nightmare Implementation

```
class Window {
   Buttons[] myButtons;
   Menus[] myMenus;
   TextAreas[] myTextAreas;
   WidgetContainer[] myContainers;

   public void update() {
      if ( myButtons != null )
         for ( int k = 0; k < myButtons.length(); k++ )
            myButtons[k].refresh();
      if ( myMenus != null )
         for ( int k = 0; k < myMenus.length(); k++ )
            myMenus[k].display();
      if ( myTextAreas != null )
         for ( int k = 0; k < myButtons.length(); k++ )
            myTextAreas[k].refresh();
      if ( myContainers != null )
         for (int k = 0; k < myContainers.length();k++)
            myContainers[k].updateElements();
      // ...etc. }
```

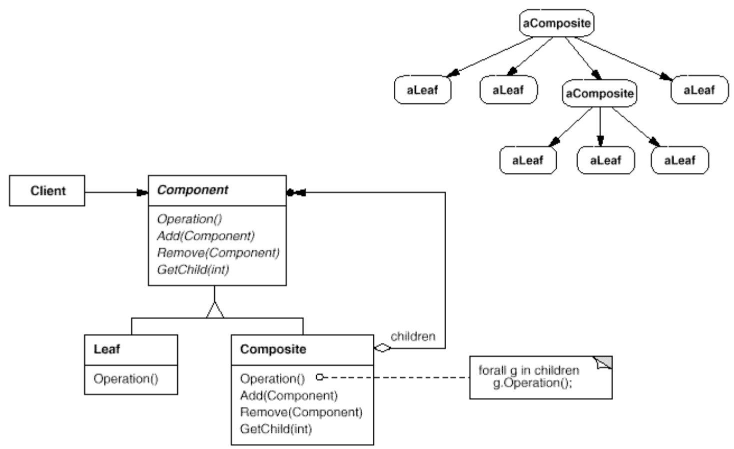

"Program to an Interface"

```
class Window {
   GUIWidgets[] myWidgets;
   WidgetContainer[] myContainers;

   public void update() {
     if(myWidgets != null)
       for (int k = 0; k < myWidgets.length(); k++)
          myWidgets[k].update();
     if(myContainers != null)
       for (int k = 0; k < myContainers.length(); k++)
          myContainers[k].updateElements();
      // .. .. etc.
   }
}
```


Basic Aspects of Composite Pattern

- Intent
  - Treat individual objects and compositions of these object uniformly
  - Compose objects into tree-structures to represent recursive aggregations
- Applicability
  - represent part-whole hierarchies of objects
  - be able to ignore the difference between compositions of objects and individual objects

Structure



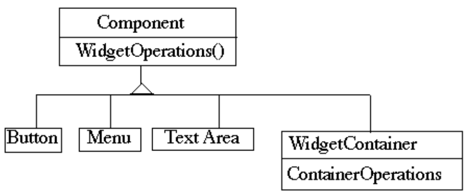
Participants & Collaborations

- Component
  - declares interface for objects in the composition
  - implements default behavior for components when possible
- Composite
  - defines behavior for components having children
  - stores child components
    - implement child-specific operations
- Leaf
  - defines behavior for primitive objects in the composition
- Client
  - manipulates objects in the composition through the Component

Consequences

- Defines uniform class hierarchies
  - recursive composition of objects
- Make clients simple
  - don't know whether dealing with a leaf or a composite
  - simplifies code because it avoids to deal in a different manner with each class
- Easier to extend
  - easy to add new Composite or Leave classes
  - glorious application of Open-Closed Principle ;)
- Design excessively general
  - type checks needed to restrict the types admitted in a particular composite structure

Applying Composite to Widget Problem



- See code
  - Component implements default behavior when possible
    - Button, Menu, etc override Component methods when needed
  - WidgetContainer will have to override all widget operations

Composite for Widgets...

```
class WidgetContainer {
   Component[] myComponents;

   public void update() {
     if ( myComponents != null )
       for( int k = 0; k < myComponents.length(); k++ )
          myComponents[k].update();
     }
}
```

Where to Place Container Operations ?

- adding, deleting, managing components in composite
  - should they be placed in Component or in Composite?
- Pro-Transparency Approach
  - Declaring them in the Component gives all subclasses the same interface
    - All subclasses can be treated alike.
  - costs safety
    - clients may do stupid things like adding objects to leaves
    - getComposite() to improve safety.
- Pro-Safety Approach
  - Declaring them in Composite is safer
    - Adding or removing widgets to non-WidgetContainers is an error

GetComposite Solution

```
class Component {
    public Composite GetComposite() { return 0; }
    //...
}

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

class Leaf extends Component { /* ... */ }

Composite aComposite = new Composite();
Leaf aLeaf = new Leaf();
Component aComponent; Composite test;

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

aComponent = aLeaf; test = aComponent->GetComposite();
if (test != null ) { test->Add(new Leaf); } // no add !
```


Other Implementation Issues

- Explicit parent references
  - simplifies traversal
  - place it in Component
  - the consistency issue
    - change parent reference **only** when add or remove child
- Child Ordering
  - consider using Iterator
- Who should delete components?
  - Composite should delete its children
- Caching to improve performance
  - cache information about children in parents

Chain of Responsibility Pattern

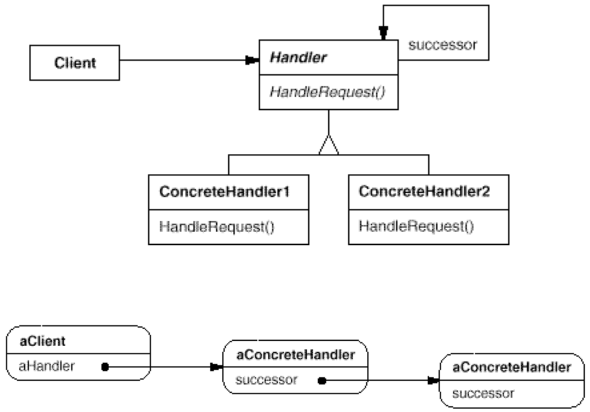
Basic Aspects

- Intent
  - Decouple sender of request from its receiver
    - by giving more than one object a chance to handle the request
  - Put receivers in a chain and pass the request along the chain
    - until an object handles it
- Motivation
  - context-sensitive help
    - a help request is handled by one of several UI objects
  - Which one?
    - depends on the context
  - The object that initiates the request does not know the object that will eventually provide the help

When to Use?

- Applicability
  - more than one object many handle a request
    - and handler isn't known a priori
  - set of objects that can handle the request should be dynamically specifiable
  - send a request to several objects without specifying the receiver

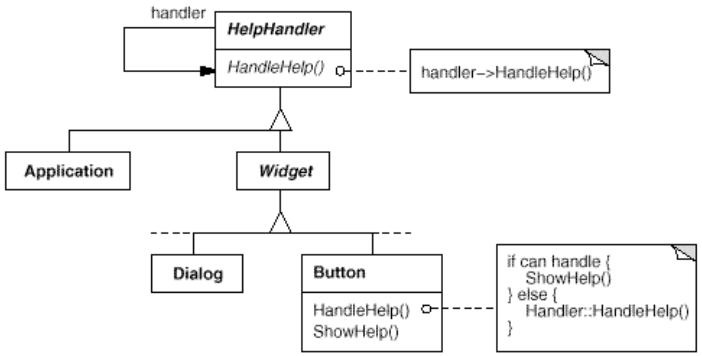
Structure



Participants & Collaborations

- Handler
  - defines the interface for handling requests
  - may implement the successor link
- ConcreteHandler
  - either handles the request it is responsible for ...
    - if possible
  - ... or otherwise it forwards the request to its successor
- Client
  - initiates the request to a ConcreteHandler object in the chain

The Context-Help System

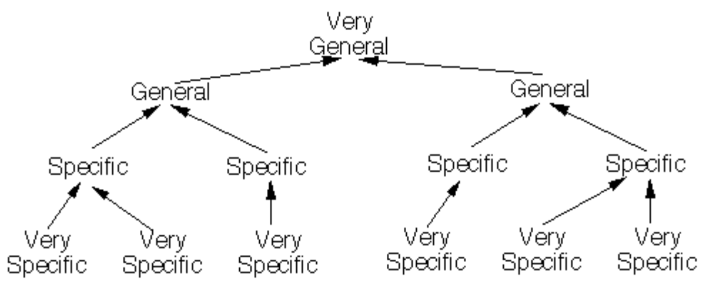


Consequences

- Reduced Coupling
  - frees the client (sender) from knowing who will handle its request
  - sender and receiver don't know each other
  - instead of sender knowing all potential receivers, just keep a single reference to next handler in chain.
    - simplify object interconnections
- Flexibility in assigning responsibilities to objects
  - responsibilities can be added or changed
  - chain can be modified at run-time
- Requests can go unhandled
  - chain may be configured improperly

How to Design Chains of Commands?

- Like the military
  - a request is made
  - it goes up the chain of command until someone has the authority to answer the request



Implementing the Successor Chain

- Define new link
  - Give each handler a link to its successor
- Use existing links
  - concrete handlers may already have pointers to their successors
    - so just use them!
  - parent references in a part-whole hierarchy
    - can define a part's successor
  - spares work and space ...
  - ... but it must reflect the chain of responsibilities that is needed

Connecting Successors

... if there are no pre-existing references for building the chain

- Successor link usually managed by Handler
  - default implementation
    - just forwards request to successor
    - frees uninterested ConcreteHandler's to implement request handling
- Sample Implementation (C++)

```
class HelpHandler {
  public:
            HelpHandler(HelpHandler* s) : successor(s) { }
     virtual void HandleHelp();
  private: HelpHandler* _successor;
};
void HelpHandler::HandleHelp () {
  if (_successor) _successor->HandleHelp();
}
```


Representing Multiple Requests using One Chain

- Each request is hard-coded
  - convenient and safe
  - not flexible
    - limited to the fixed set of requests defined by handler
- Unique handler with parameters
  - more flexible
  - but it requires conditional statements for dispatching request
  - less type-safe to pass parameters
- Unique handler with Request object parameter
  - subclasses extend rather than overwrite the handler method

Multiple Requests - Solution 1: Hard-Coded

```
abstract class HardCodedHandler {
   private HardCodedHandler successor;

   public HardCodedHandler( HardCodedHandler aSuccessor)
      { successor = aSuccessor; }

   public void handleOpen()
      { successor.handleOpen(); }

   public void handleClose()
      { successor.handleClose(); }

   public void handleNew( String fileName)
      { successor.handleNew( fileName ); }
}
```


Multiple Requests - Solution 2: Unique Parameterized Handle

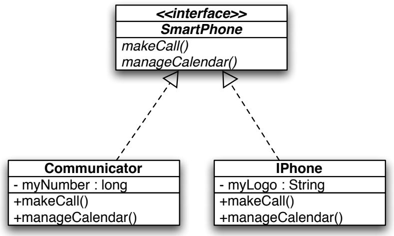
```
abstract class SingleHandler {
   private SingleHandler successor;

   public SingleHandler( SingleHandler aSuccessor) {
      successor = aSuccessor;
   }

   public void handle( String request) {
      successor.handle( request );
   }
}

class ConcreteOpenHandler extends SingleHandler {
   public void handle( String request) {
      switch ( request ) {
         case "Open" : // do the right thing;
         case "Close" : // more right things;
         case "New" : // even more right things;
         default: successor.handle( request );
      }
   }
}
```

Multiple Requests - Solution 3: Request Object

```
void Handler::HandleRequest (Request* theRequest) {
    switch (theRequest->GetKind()) {
      case Open: HandleOpen((OpenRequest*) theRequest); break;
      case New:  HandleNew((NewRequest*) theRequest);
         /* ... */ break;
      default: /* ... */ break;
    } }

class ExtendedHandler : public Handler {
    public: virtual void HandleRequest(Request* theRequest);
     // ... };

void ExtendedHandler::HandleRequest(Request* r) {
    switch (r ->GetKind()) {
      case Preview:
        // handle the Preview request
        break;

      default:
        // let Handler handle other requests
        Handler::HandleRequest(r);
    } }
```

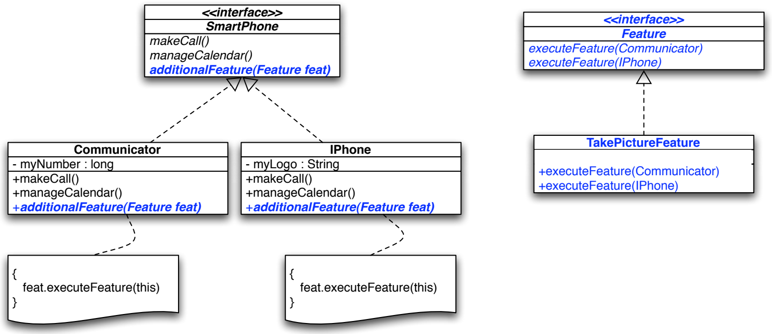

Let's Play with Smart Phones...

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



Implementation Options

- Is one **executeFeature()** method enough?
  - we need TWO "containers" for the two distinct implementations
    - one method per type of phone
  - one method with a switch... phew! :(
- 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)**
  - switch stays in one place...
    - independently on the number of new features

Double Dispatch

- Actually what we have is a bi-dimensional matrix of features:

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

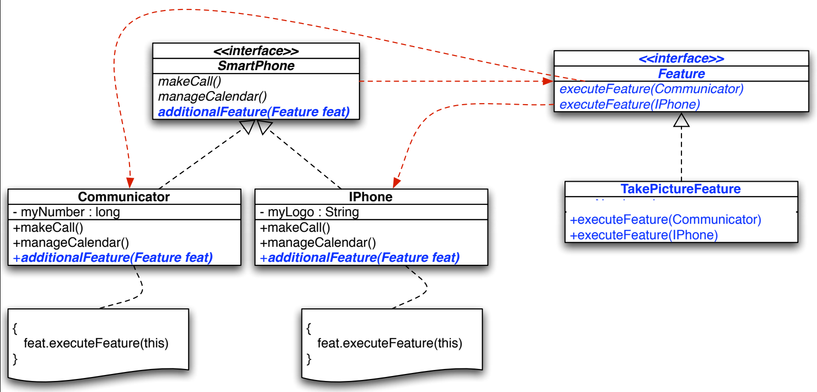
- Actually what we have is a bi-dimensional matrix of features:

The Matrix Reveals a Problem...

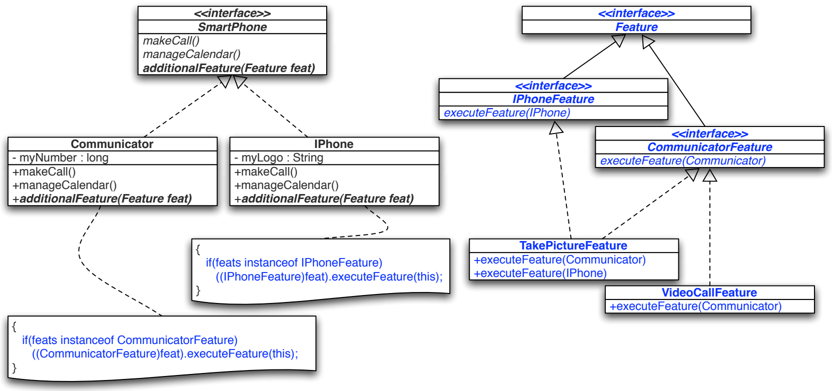
- It is easy add a new **Feature**, but hard to add a new **SmartPhone**
  - We have to change the entire **Feature** hierarchy!!
- ...and even if we change who says that all SmartPhone will have all the additional features?!!
- In other words:

WHAT IF THE MATRIX IS SPARSE?

The True Problem: Cyclic Dependencies



Second Solution: Remove Cycles



Visitor

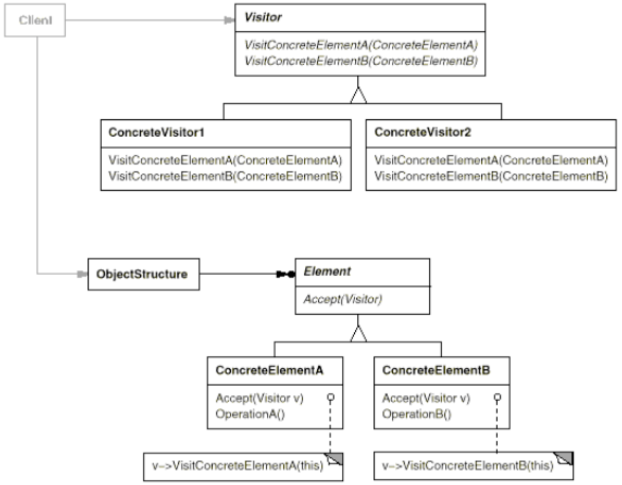
Visitor

- 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

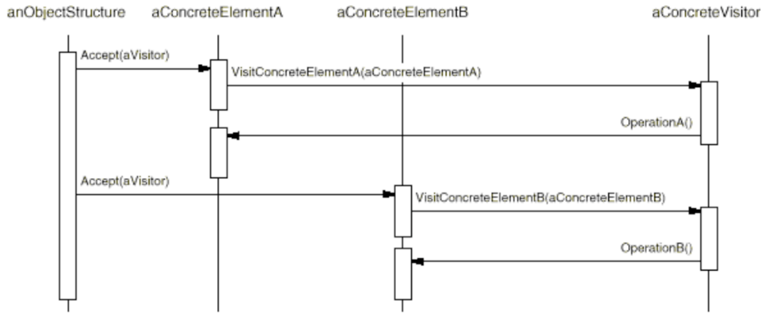
You want to use it when...

- 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

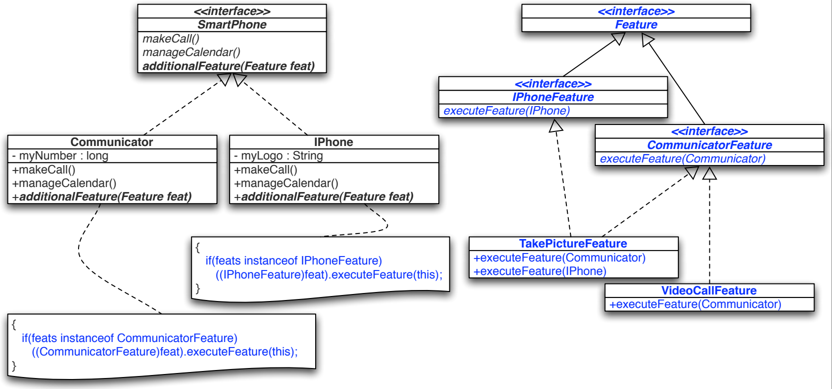
Issue of Cyclic Dependencies

- 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**!

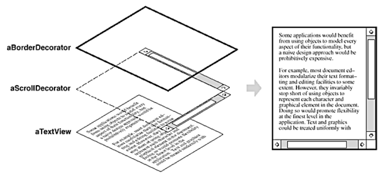
Acyclic Visitor on Example



A Class Inflation Problem...

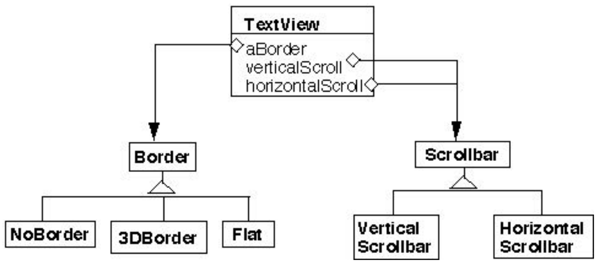
Motivation

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



- 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

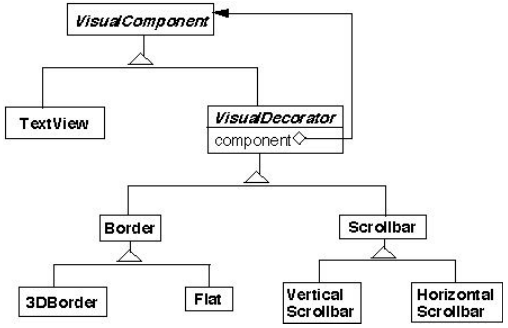


- Is it Open-Closed?

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.
}
```

Solution 2: Change the Skin, not the Guts!



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

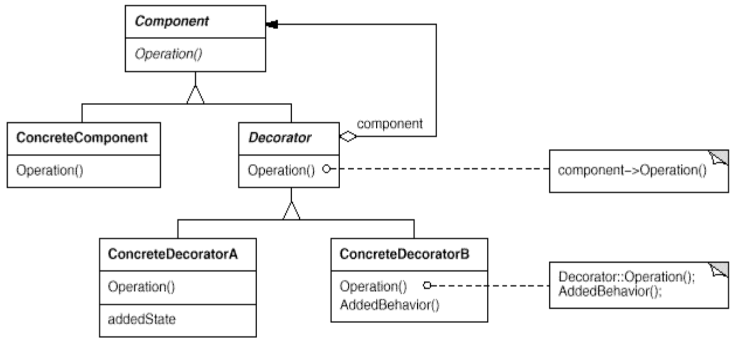
Decorator Pattern

Changing the skin of an object

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

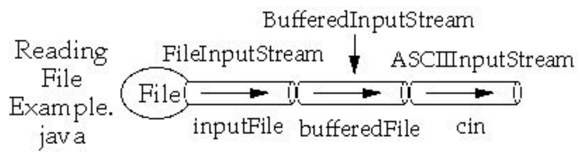
Consequences

- 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

Decorator Example from Java API



Source Code for Java API Example

```
import java.io.*;

class  ReadingFileExample {
  public  static  void  main( String  args[]  )
  throws Exception {
      FileInputStream inputFile;
      BufferedInputStream bufferedFile;
      ASCIIInputStream  cin;

     inputFile = new FileInputStream("ReadFileEx.java");
     bufferedFile = new BufferedInputStream(inputFile );
     cin = new ASCIIInputStream( bufferedFile );

     System.out.println(  cin.readWord()  );
     for   ( int   k = 0 ;   k   <   4;   k++ )
       System.out.println(  cin.readLine()   );
 }
}
```


Decorator vs. Chain of Responsibility

Chain of Responsibility	Decorator
Comparable to "event-oriented" architecture	Comparable to layered architecture (layers of an onion)
The "filter" objects are of equal rank	A "core" object is assumed, all "layer" objects are optional
User views the chain as a "launch and leave" pipeline	User views the decorated object as an enhanced object
A request is routinely forwarded until a single filter object handles it. many (or all) filter objects *could* contrib. to each request's handling.	A layer object always performs pre or post processing as the request is delegated.
All the handlers are peers (like nodes in a linked list) – "end of list" condition handling is required.	All the layer objects ultimately delegate to a single core object - "end of list" condition handling is not required.