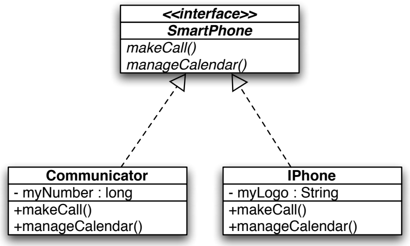
ADDING Dynamically Behavior to Objects

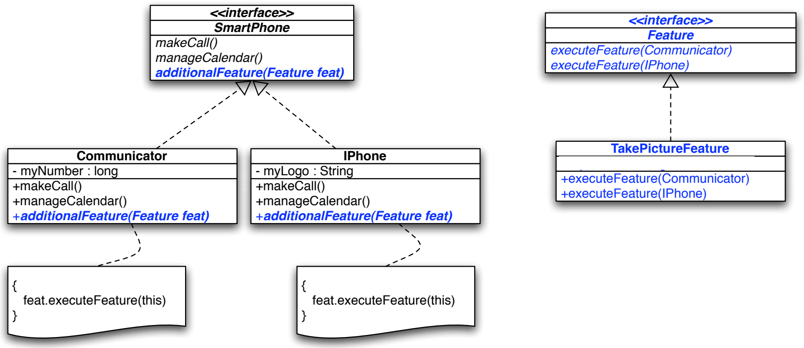
Let's Play with Smart Phones...

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

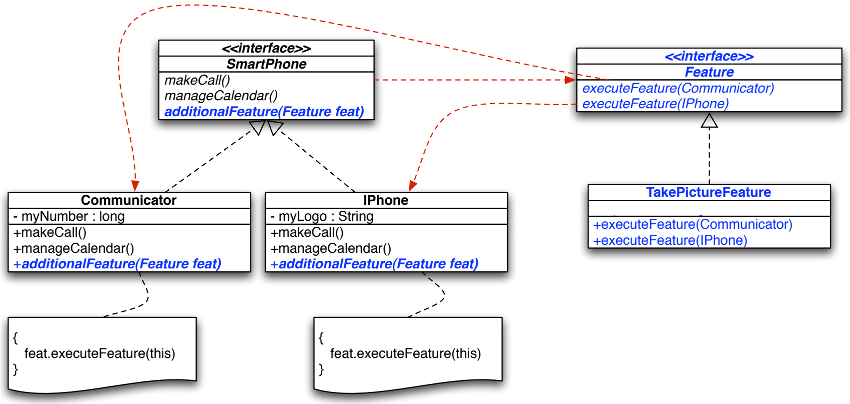
	Features			
Smart Phones		Take Pictures	Video Call	....
	IPhone	X	X	
	Communicator	X	X	
	....			

The Matrix Reveals a Problem...

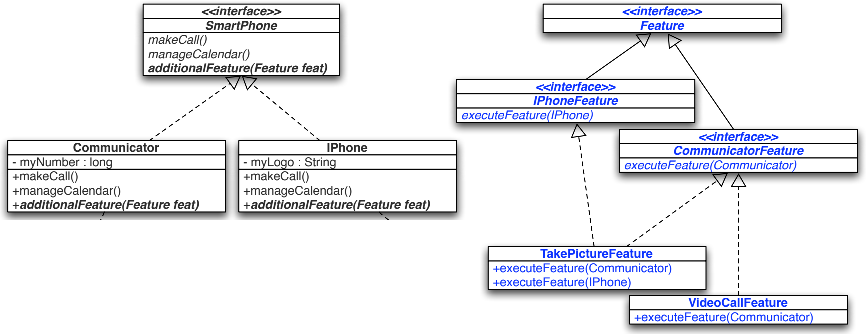
- easy to add a new `Feature`, but hard to add a new `SmartPhone`
  - We have to change the entire `Feature` hierarchy!!
- ...and even if we change, who says that all SmartPhone will have all the additional features?!!
- In other words:

WHAT IF THE MATRIX IS SPARSE?

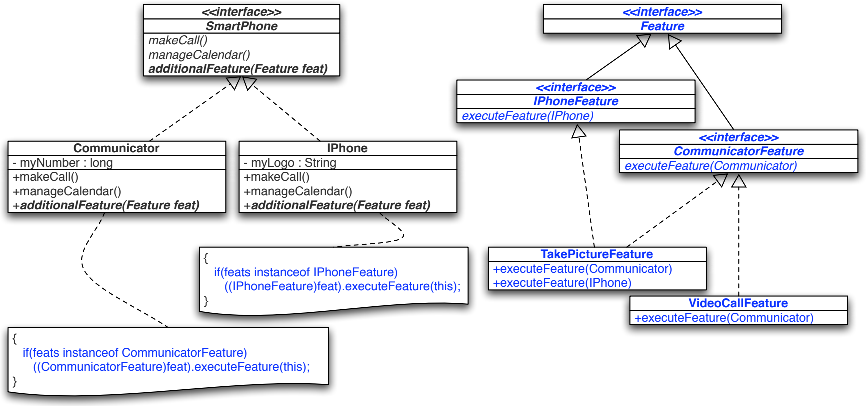
Cyclic Dependencies



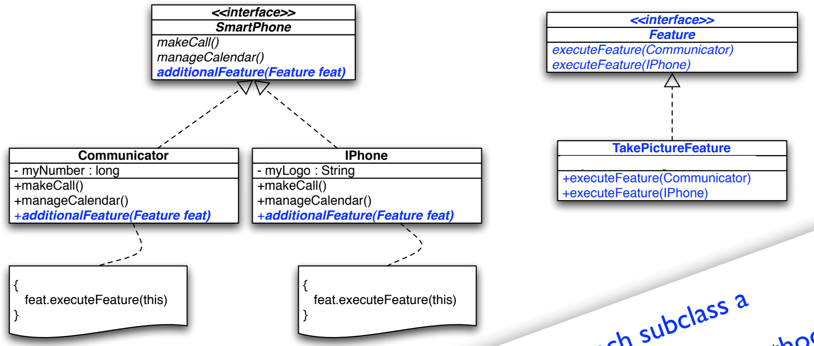
Second Solution: Remove Cycles



Second Solution: Remove Cycles

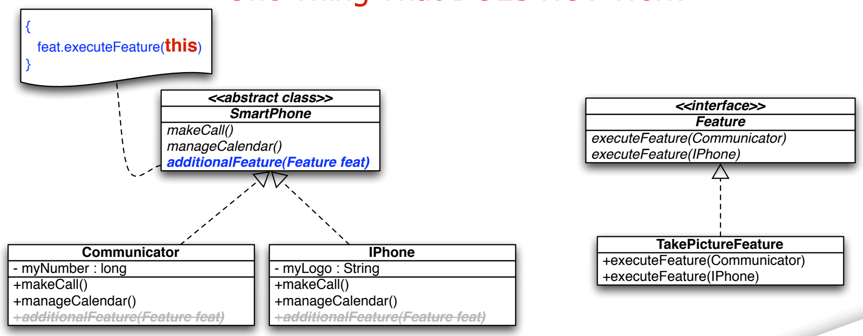


First Solution Revisited



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

One Thing That DOES NOT Work



The method executeFeature(Communicator) in the type Feature is not applicable for the arguments (SmartPhone)

This is the limitation of single dispatch! (polymorphism works only for caller object)

You can "patch" it... but it's not a good idea

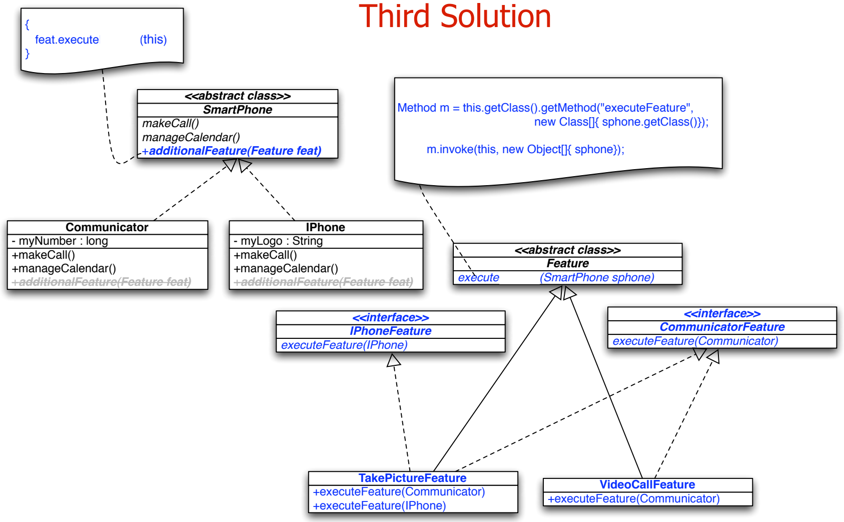
```
abstract class Feature {
    protected abstract void executeFeature(IPhone anIPhone);
    protected abstract void executeFeature(Communicator aCommunicator);
    public void executeFeature(SmartPhone sphone) {
        if(sphone instanceof Communicator) executeFeature((Communicator)sphone);
        else if(sphone instanceof IPhone) executeFeature((IPhone)sphone);
    }
}
```

Double Dispatch by Reflection

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

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

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

1 2

Third Solution



Visitor

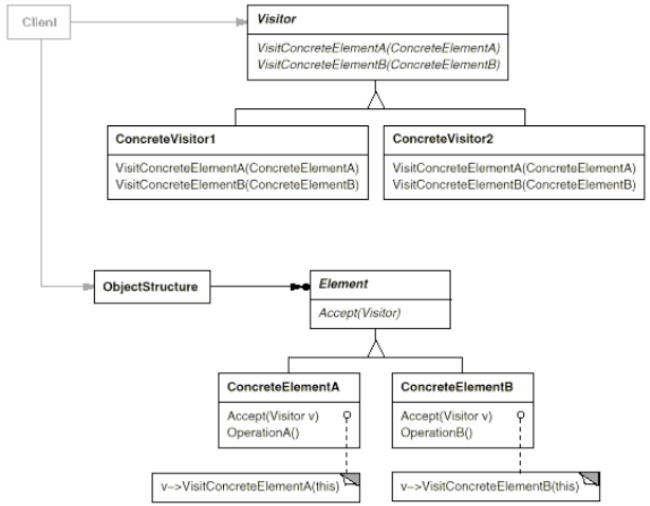
Visitor

- 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

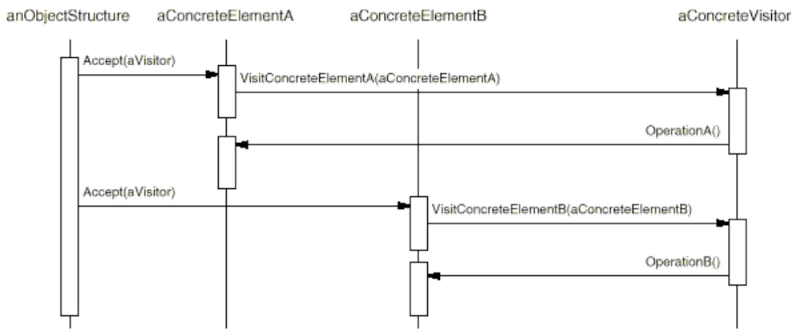
You want to use it when...

- 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.

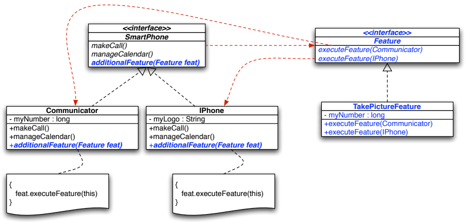
- The visitor is used least often to do it, because traversal code often gets duplicated.

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

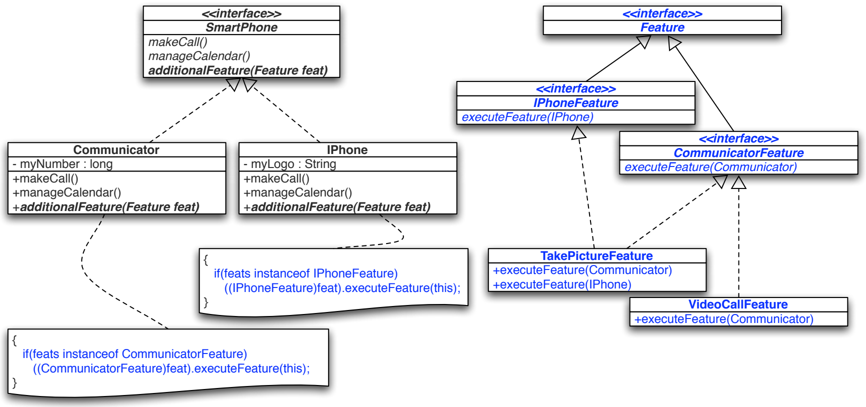
Acyclic Visitor

- 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

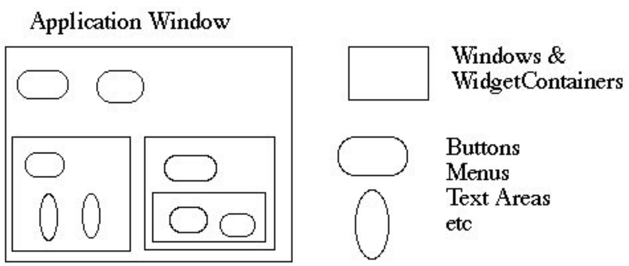
- **Acyclic Visitor** is like a **sparse matrix**!

Acyclic Visitor on Example



Composite Pattern

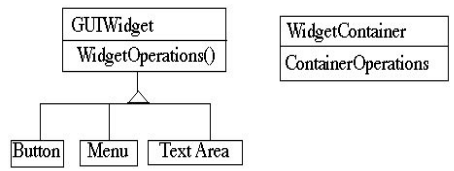
Motivation



- 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

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



1. 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. }
```

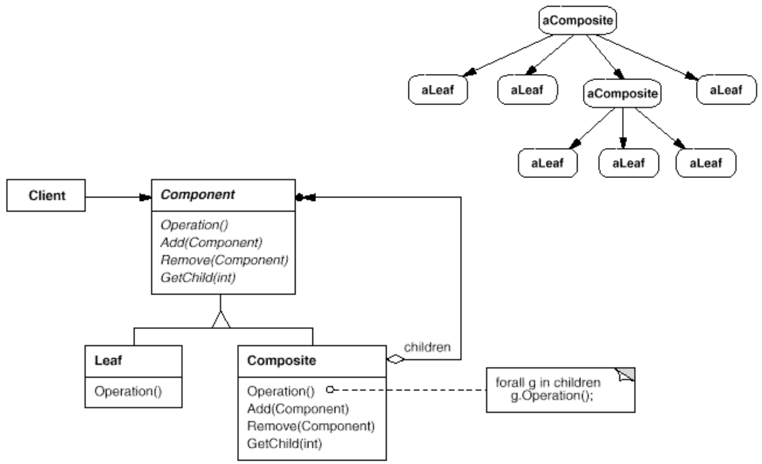

2. "Program to an Interface"

```
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



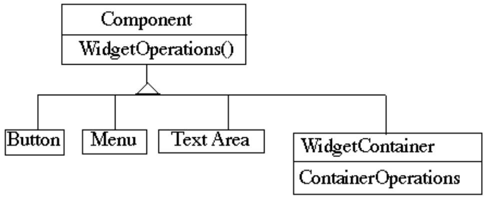
Participants & Collaborations

- 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 interface

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

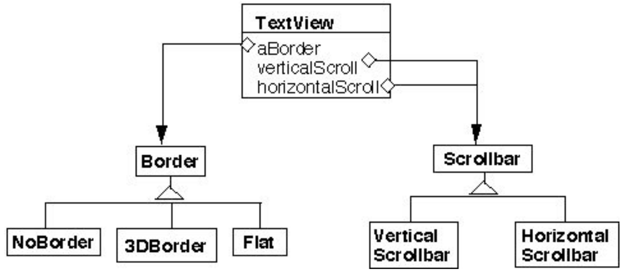
```
class Component {
    public Composite GetComposite() { return 0; }
    //...
}

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

A Class Inflation Problem...

Solution 1: Use Object Composition

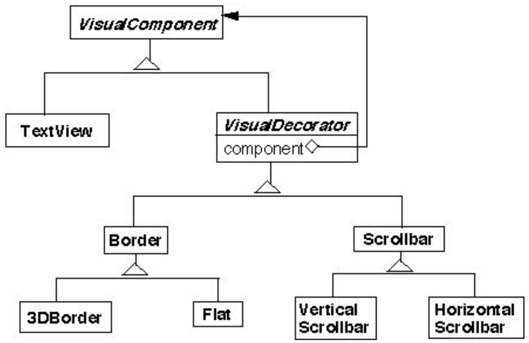


Solution 1: The Source-Code

```
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!



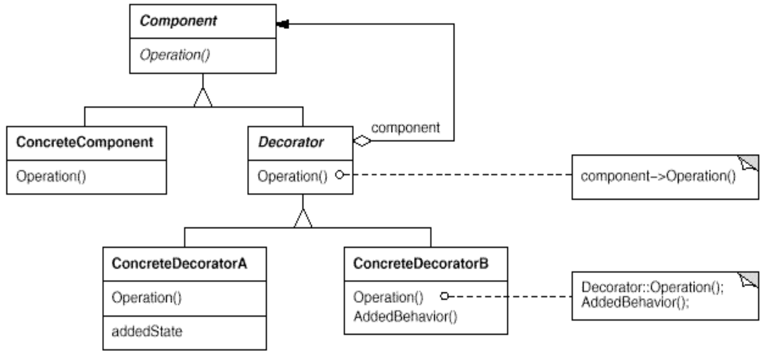
- TextView has **no** borders or scrollbars!
- Add borders and scrollbars on top of a TextView

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

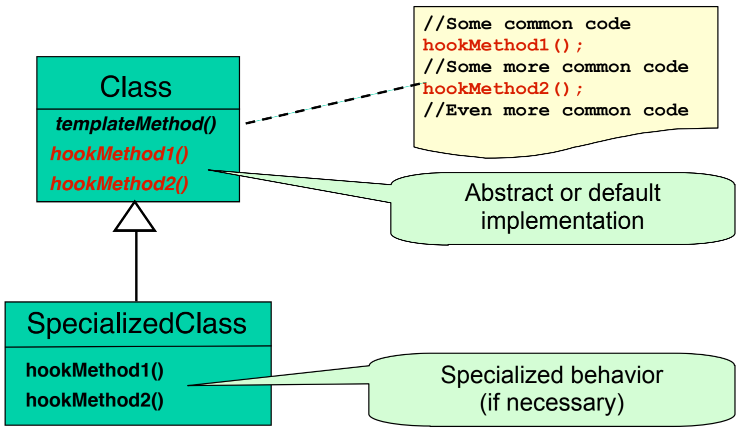
- More flexibility than static inheritance
  - allows to mix and match responsibilities
  - allows to apply a property twice
- Avoid feature-laden classes high-up in the hierarchy
  - "pay-as-you-go" approach
  - easy to define new types of decorations
- Lots of little objects
  - easy to customize, 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

Template Method vs. Strategy

Remember the **Template Method** Pattern...

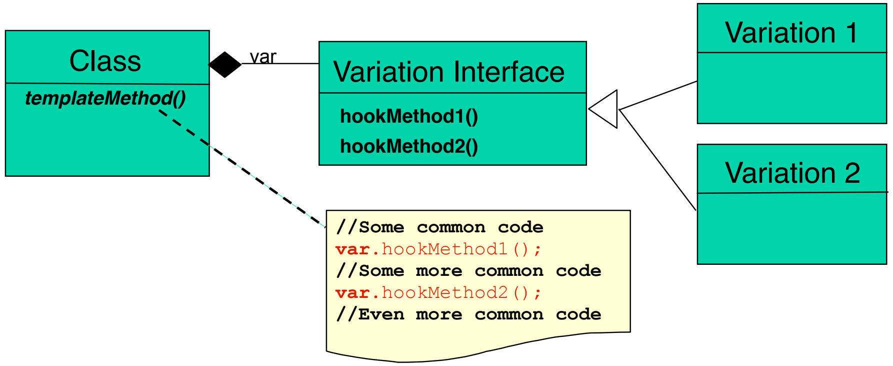


What does it really mean?

- One “algorithm” (i.e. program logic) with many variations....
- The idea is to
  - put the algorithm in **one place** and
  - make variation points explicit ...
  - ...and then let them be re-implemented by subclasses
- It’s cool, but it’s static
  - I can’t change my algorithm dynamically (at run-time)
- What should I do?

Favor Composition over Inheritance!

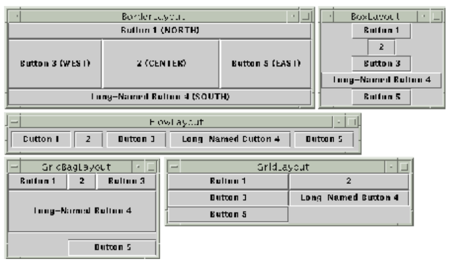
Solution based on Composition



Strategy Pattern

Java Layout Managers

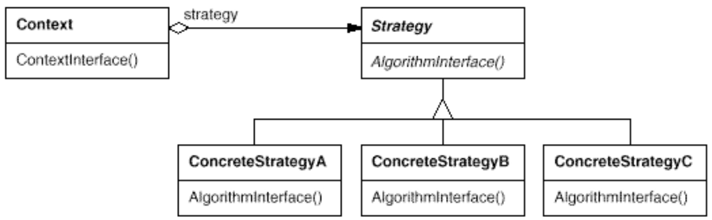
- GUI container classes in Java
  - frames, dialogs, applets (top-level)
  - panels (intermediate)
- Each container class has a layout manager
  - determine the size and position of components
  - 20 types of layouts
  - ~40 container-types
  - imagine to combine them freely by inheritance ;)
- Consider also sorting...
  - open-ended number of sorting criteria



Basic Aspects

- Intent
  - Define a family of algorithms, encapsulate each one, and make them interchangeable
  - Let the algorithm vary independently from clients that use it
- Applicability
  - You need different variants of an algorithm
  - An algorithm uses data that clients shouldn't know about
    - avoid exposing complex, algorithm-specific data structures
  - Many related classes differ only in their behavior
    - configure a class with a particular behavior

Structure



Participants

- Strategy
  - declares an interface common to all supported algorithms.
  - Context uses this interface to call the algorithm defined by a ConcreteStrategy
- ConcreteStrategy
  - implements the algorithm using the Strategy interface
- Context
  - configured with a ConcreteStrategy object
  - may define an interface that lets Strategy objects to access its data

Positive Consequences

- Families of related algorithms
  - usually provide different implementations of the same behavior
  - choice decided by time vs. space trade-offs
- Alternative to subclassing
  - We still subclass the strategies...Why is this a big deal? ;)
- Eliminates conditional statements
  - many conditional statements → "invitation" to apply Strategy!

Negative Consequences

- Communication overhead between Strategy and Context
  - some ConcreteStrategies don't need information passed from Context
- Clients must be aware of different strategies
  - clients must understand the different strategies

```
SortedList studentRecords = new SortedList(new ShellSort());
```

- Increased number of objects
  - each Context uses its concrete strategy objects
  - can be reduced by keeping strategies stateless (share them)

Implementation

- How does data flow between Context and Strategies?
  - **Approach 1:** take data to the strategy
    - decoupled, but might be inefficient
  - **Approach 2:** pass Context itself and let strategies take data
    - Context must provide a more comprehensive access to its data
    - **more coupled**
  - In Java strategy hierarchy might be inner classes
- Making Strategy object optional
  - provide Context with default behavior
    - if default used no need to create Strategy object
  - don't have to deal with Strategy unless you don't like the default behavior

Chain of Responsibility Pattern

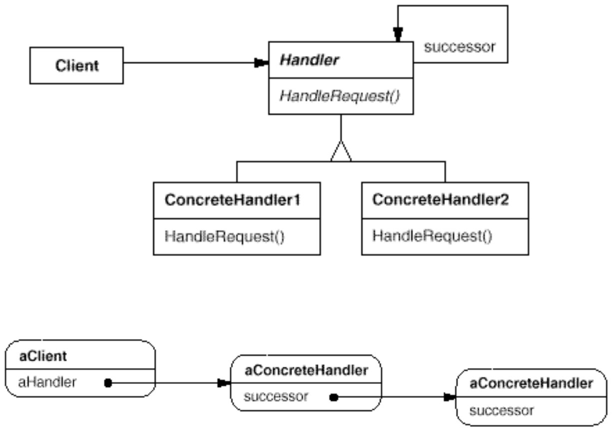
Basic Aspects

- 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

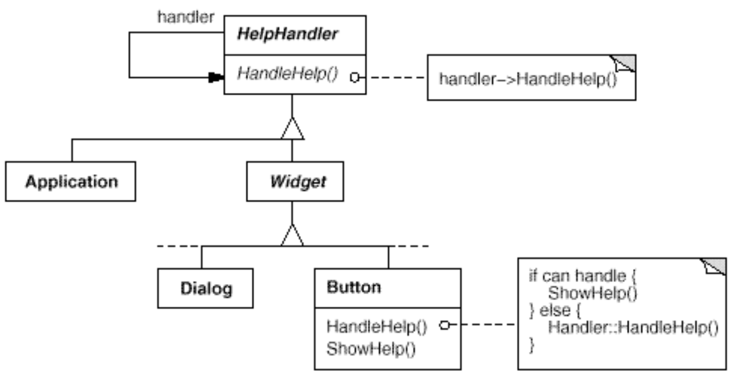
Structure



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

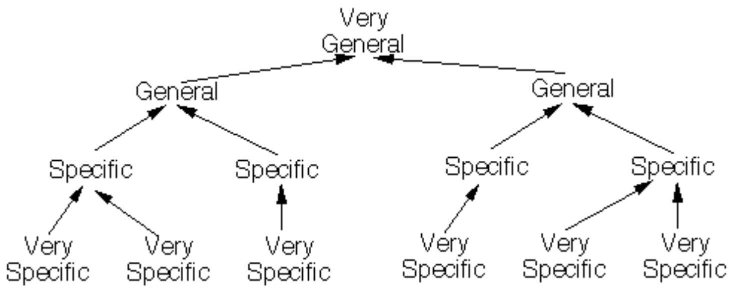


Consequences

- 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

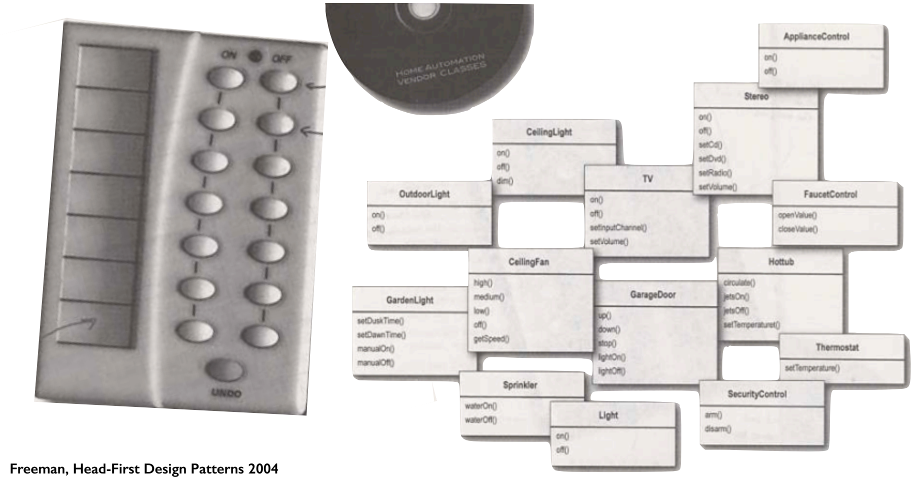
Connecting Successors

... 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();
}
```


Encapsulating Invocations



Freeman, Head-First Design Patterns 2004

How Do I Avoid Hard-Coding Devices to Slots?

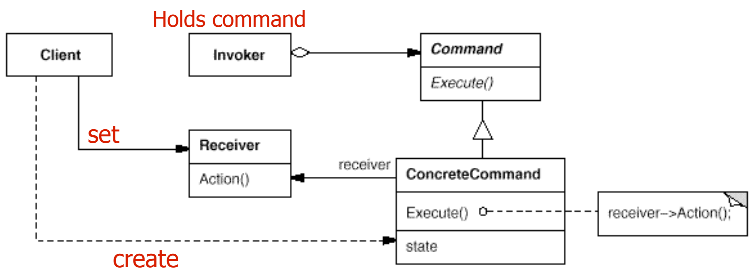
- Sometimes all you know is that calling a method needs to trigger an action... but you can't know what action
- Sometimes you need to organize actions
  - e.g. group them in collections, run statistics
- Sometimes you need to "record" ("backup") actions
  - to trace a symptom... or to restore a system

Treat Actions as Objects!

Command Pattern

"Objectifying" Actions

Structure



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

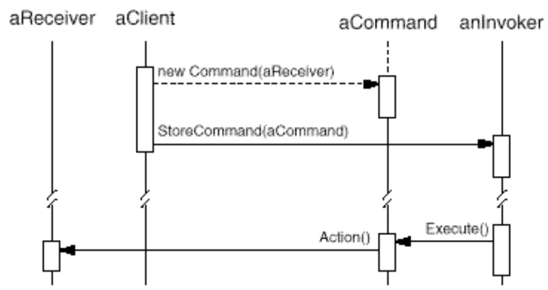
Basic Aspects

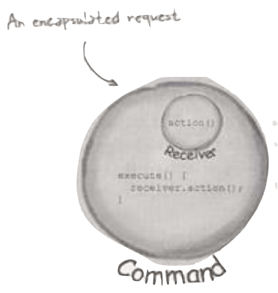
- Intent
  - Encapsulate requests as objects, letting you to:
    - parameterize clients with different requests
    - queue or log requests
    - support undoable operations
- Applicability
  - Parameterize objects
    - replacement for callbacks
  - Specify, queue, and execute requests at different 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

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

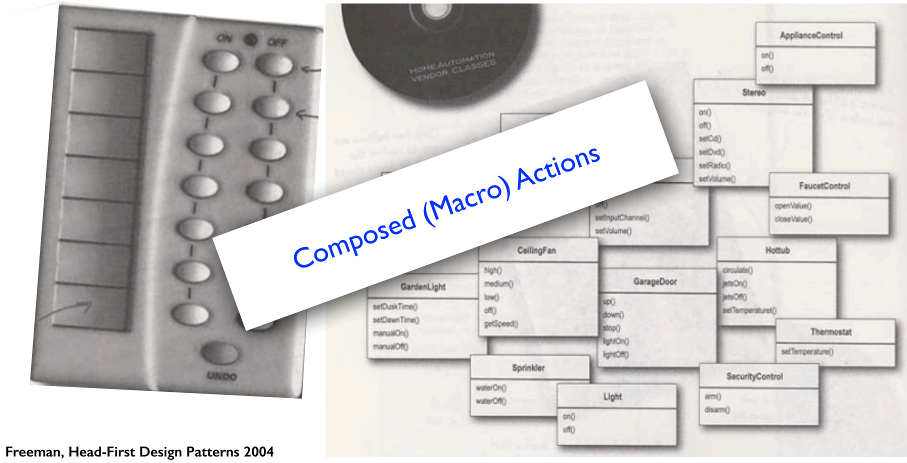
Intelligence of Command objects

- "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

What If we want to Define Activities?



Freeman, Head-First Design Patterns 2004

Composed Commands