

Design Patterns

CMSC 433

What is a pattern?

- Patterns = problem/solution pairs in context
- Patterns facilitate reuse of successful software architectures and design
- Not code reuse
 - Instead, solution/strategy reuse
 - Sometimes, interface reuse

Design patterns, CMSC 433

2

Gang of Four

- The book that started it all
- Community refers to authors as the “Gang of Four”
- Figures and some text in these slides come from book



Design patterns, CMSC 433

3

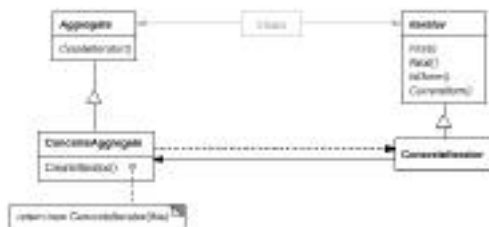
Some design patterns you already know

- Iterator
 - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Observer
 - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Proxy
 - Provide a surrogate or placeholder for another object to control access to it.

Design patterns, CMSC 433

4

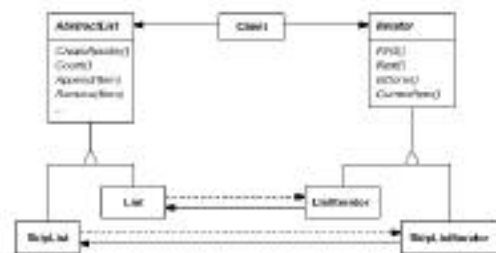
Iterator pattern



Design patterns, CMSC 433

5

Uses of Iterator Pattern



Design patterns, CMSC 433

6

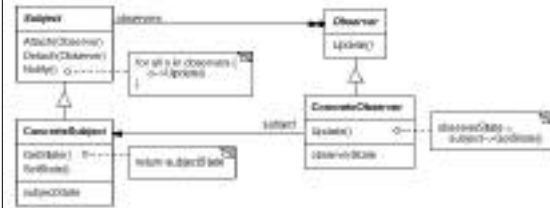
You've already seen this in Java

- Iterators for Collections
 - Also, Enumerators for Hashtables and Vectors

Design patterns, CMSC 433

7

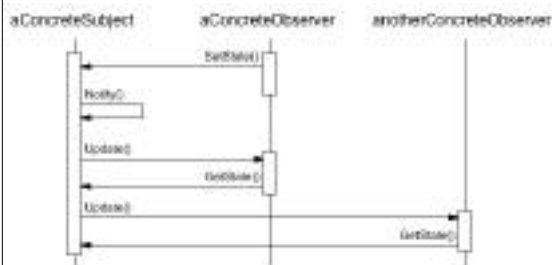
Observer pattern



Design patterns, CMSC 433

8

Use of an Observer pattern



Implementation details

- Observing more than one subject.
 - It might make sense in some situations for an observer to depend on more than one subject. The subject can simply pass itself as a parameter in the Update operation, thereby letting the observer know which subject to examine.
- Making sure Subject state is self-consistent before notification.

Design patterns, CMSC 433

10

More Implementation Issues

- Implementations of the Observer pattern often have the subject broadcast additional information about the change.
 - At one extreme, the subject sends observers detailed information about the change, whether they want it or not. At the other extreme the subject sends nothing but the most minimal notification, and observers ask for details explicitly thereafter
- You can extend the subject's registration interface to allow registering observers only for specific events of interest.

Design patterns, CMSC 433

11

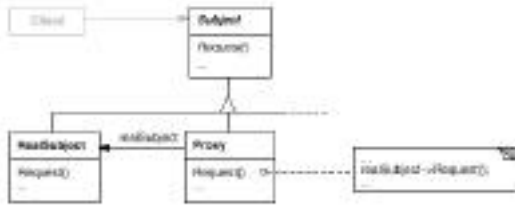
You've seen this before

- The standard Java and JavaBean event model is an example of an observer pattern

Design patterns, CMSC 433

12

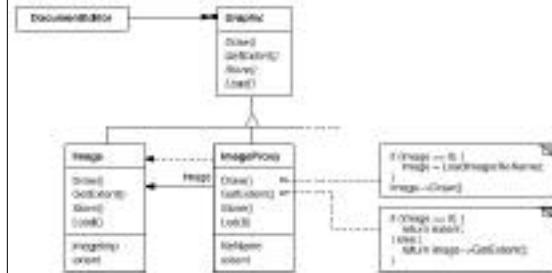
Proxy pattern



Design patterns, CMSC 433

13

Example of Proxy Pattern



Design patterns, CMSC 433

14

Known uses (from DP book)

- McCullough [McC87] discusses using proxies in Smalltalk to access remote objects. Pascoe [Pas86] describes how to provide side-effects on method calls and access control with "Encapsulators."
- NEXTSTEP [Add94] uses proxies (instances of class NXProxy) as local representatives for objects that may be distributed.
 - On receiving a message, the proxy encodes it along with its arguments and then forwards the encoded message to the remote subject. Similarly, the subject encodes any return results and sends them back to the NXProxy object.

Design patterns, CMSC 433

15

Some new patterns

Creation patterns

- Singleton
 - Ensure a class only has one instance, and provide a global point of access to it.
- Abstract Factory
 - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Design patterns, CMSC 433

17

Structural patterns

- Adapter
 - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Design patterns, CMSC 433

18

Behavioral patterns

- State
 - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Visitor
 - Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Design patterns, CMSC 433

19

Singleton pattern

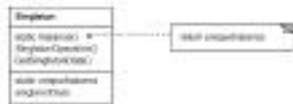
- It's important for some classes to have exactly one instance
 - Many printers, but only one print spooler
 - One file system
 - One window manager
- Such designs can be limiting

Design patterns, CMSC 433

20

The Singleton solution

- Make the class itself responsible for keeping track of its sole instance.
- Make constructor private
- Provide static method/field to allow access to the only instance of the class



Design patterns, CMSC 433

21

Implementing the Singleton method

- In Java, just define a final static field

```
Class Singleton {
  private Singleton() {...}
  public final static Singleton instance
    = new Singleton();
  ...
}
```
- Java semantics guarantee object is created immediately before first use

Design patterns, CMSC 433

22

Implementing the Singleton Method in C++

- C++ static initialization is very limited and ill-defined
- Use instance() method
- Note: not thread safe
 - Java method is thread safe

Design patterns, CMSC 433

23

Implementing the Singleton Method in C++

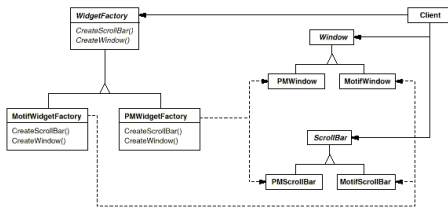
```
Class Singleton {
  private: Singleton() {...}
  static Singleton _instance = 0;
  public: static Singleton instance() {
    if (_instance == 0)
      _instance = new Singleton();
    return _instance;
  }
  ...
}
```

Design patterns, CMSC 433

24

Abstract Factory

- Different look-and-feels define different appearances and behaviors for user interface “widgets” like scroll bars, windows, and buttons.



Using an abstract factory

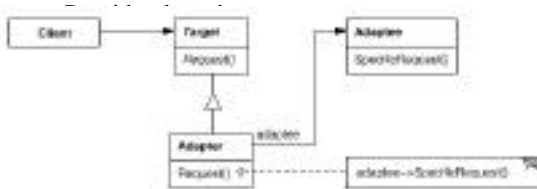
- Get a reference to of type WidgetFactory
 - To an object of the appropriate subtype of WidgetFactory
 - Ask the WidgetFactory for a scroll bar, or for a window

Design patterns, CMSC 433

26

Adapter pattern

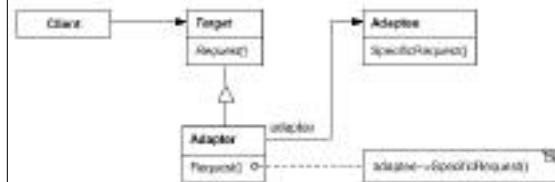
- Clients needs a target that implements one interface



Design patterns, CMSC 433

27

Structure of Adapter pattern



Design patterns, CMSC 433

28

JDK 1.3 Proxy class

- Can be used for both Proxy and Adaptor pattern
- Use java.lang.reflect.Proxy
- Create object that implements a set of interfaces
 - Specified dynamically
 - Invocation handler handles calls

Design patterns, CMSC 433

29

State pattern

- Suppose an object is always in one of several known states
- The state an object is in determines the behavior of several methods
- Could use if/case statements in each method
- Better solution: state pattern

Design patterns, CMSC 433

30

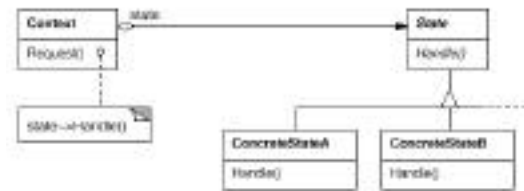
State pattern

- Have a reference to a state object
 - Normally, state object doesn't contain any fields
 - Change state: change state object
 - Methods delegate to state object

Design patterns, CMSC 433

31

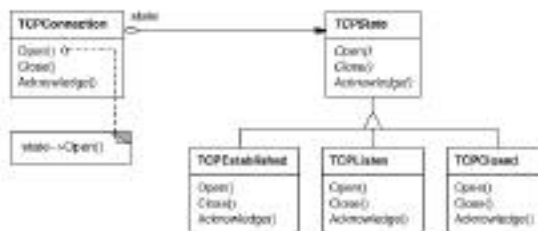
Structure of State pattern



Design patterns, CMSC 433

32

Instance of State Pattern



Design patterns, CMSC 433

33

State pattern notes

- Can use singletons for instances of each state class
 - State objects don't encapsulate state, so can be shared
- Or use inner classes
 - Each state object contains reference to outer object
- Easy to add new states
 - New states can extend other states
 - Override only selected functions

Design patterns, CMSC 433

34

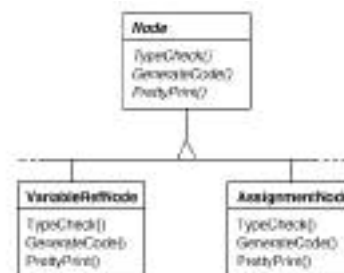
Visitor pattern

- A visitor encapsulates the operations to be performed on an entire structure
 - E.g., all elements of a parse tree
- Allows them to be specified separately from the parse tree
 - But doesn't require putting all of the structure traversal code into each visitor/operation

Design patterns, CMSC 433

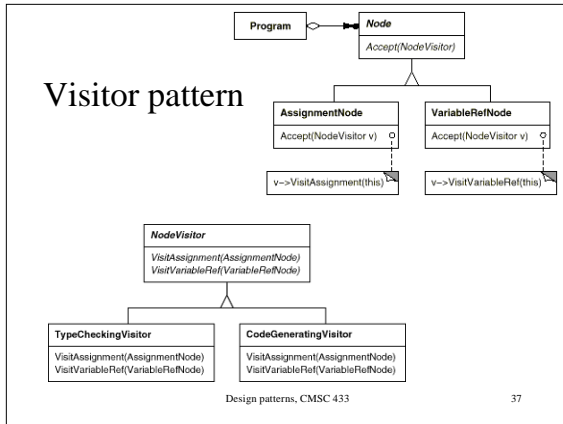
35

Not a visitor



Design patterns, CMSC 433

36



Double-dispatch

- Accept code is always trivial
 - Just dynamic dispatch on argument, with runtime type of structure node taking into account in method name
- A way of doing double-dispatch
 - Dynamic dispatching on the run-time types of two arguments
 - Visitor code invoked depends on run-time type of both visitor and node

Design patterns, CMSC 433 38

Using overloading in a visitor

- You can name all of the visitXXX(XXX x) methods just visit(XXX x)
 - Calls to Visit (AssignmentNode n) and Visit(VariableRefNode n) distinguished compile-time overload resolution

Design patterns, CMSC 433 39

Can provide default behavior

- Default visit(BinaryPlusOperatorNode) can just forward call to visit(BinaryOperatorNode)
- Visitor can just provide implementation of visit(BinaryOperatorNode) if it doesn't care what type of binary operator node it is at

Design patterns, CMSC 433 40

State in a visitor pattern

- Visitor's can contain state
 - E.g., the results of parsing the program so far
- Or use stateless visitors and pass around a separate state object

Design patterns, CMSC 433 41

Traversals

- In the standard visitor pattern, the visitor at a node is responsible for visiting the components (i.e., children) of that node
 - if that is what is desired
 - Visitors can be applied to flat object structures
- Several solutions
 - acceptAndTraverse methods
 - Visit/process methods
 - traversal visitors applying an operational visitor

Design patterns, CMSC 433 42

acceptAndTraverse methods

- accept method could be responsible for traversing children
 - Assumes all visitors have same traversal pattern
 - E.g., visit all nodes in pre-order traversal
 - Could provide previsit and postvisit methods to allow for more complicated traversal patterns
 - Still visit every node
 - Can't do out of order traversal
 - In-order traversal requires inVisit method

Design patterns, CMSC 433

43

Accept and traverse

- Class BinaryPlusOperatorNode {

```
void accept(Visitor v) {  
    v->visit(this);  
    lhs->accept(v);  
    rhs->accept(v);  
}
```

...}

Design patterns, CMSC 433

44

Visitor/process methods

- Can have two parallel sets of methods in visitors
 - Visit methods
 - Process methods
- Visit method on a node:
 - Calls process method of visitor, passing node as an argument
 - Calls accept on all children of the node (passing the visitor as an argument)

Design patterns, CMSC 433

45

Preorder visitor

- Class PreorderVisitor {

```
void visit(BinaryPlusOperatorNode n) {  
    process(n);  
    n->lhs->accept(this);  
    n->rhs->accept(this);  
}
```

...}

Design patterns, CMSC 433

46

Visit/process, continued

- Can define a PreorderVisitor
 - Extend it, and just redefine process method
 - Except for the few cases where something other than preorder traversal is required
- Can define other traversal visitors as well
 - E.g., PostOrderVisitor

Design patterns, CMSC 433

47

Traversal visitors applying an operational visitor

- Define a Preorder traversal visitor
 - Takes an operational visitor as an argument when created
- Perform preorder traversal of structure
 - At each node
 - Have node accept operational visitor
 - Have each child accept traversal visitor

Design patterns, CMSC 433

48

PreorderVisitor with payload

- Class PreorderVisitor {
 Visitor payload;
 void visit(BinaryPlusOperatorNode n) {
 payload->visit(n);
 n->lhs->accept(this);
 n->rhs->accept(this);
 }
 ...}

Design patterns, CMSC 433

49

Why patterns?

- Sometimes, patterns are just a cool idea you might not have come up with on your own
 - Learn from others
- If you work with code written by someone else
 - If they use a pattern you know, it will be easier for you to understand their code
- Some frameworks can automatically build/manipulate certain design patterns

Design patterns, CMSC 433

50

Pattern hype

- Patterns get a lot of hype and fanatical believers
 - *We are going to have a design pattern reading group, and this week we are going to discuss the Singleton Pattern!*
- Patterns are sometimes wrong (e.g., double-checked locking) or inappropriate for a particular language or environment
 - Patterns developed for C++ have very different solutions in Smalltalk or Dylan

Design patterns, CMSC 433

51