

CMSC 433 – Programming Language Technologies and Paradigms Spring 2005

Iterators and Design Patterns
February 15, 2005

1

Administrivia

- Other resources:
 - *Thinking in Patterns with Java*
 - [Link from the class web page](#)
 - Gamma, Helm, Johnson, Vlissides, *Design Patterns*, Addison-Wesley 1995
 - Freeman et al, *Head First Design Patterns*, O'Reilly, 2004

2

Inner Classes

- Classes can be nested inside other classes
 - These are called *inner classes*
- Within a class that contains an inner class, you can use the inner class just like any other class

3

Example: The Queue Class

```
class Queue<Element> {
    class Entry { // Java inner class
        Element elt; Entry next;
        Entry(Element i) { elt = i; next = null; }
    }

    private Entry theQueue;

    void enqueue(Element e) {
        if (theQueue == null) theQueue = new Entry(e);
        else {
            Entry last = theQueue;
            while (last.next != null) last = last.next;
            last.next = new Entry(e);
        }
    }
    ...
}
```

4

Example: The Queue Class (cont'd)

```
class Queue<Element> {
    ...
    Element dequeue() throws EmptyQueueException {
        if (theQueue == null)
            throw new EmptyQueueException();
        Element e = theQueue.elt;
        theQueue = theQueue.next;
        return e;
    }
}
```

5

Referring to Outer Class

```
class Queue<Element> {
    ...
    int numEntries;
    class Entry {
        Element elt; Entry next;
        Entry(Element i) { elt = i; next = null;
                          numEntries++; }
    }
}
```

- Each inner “object” has an implicit reference to the outer “object” whose method created it
 - Can refer to fields directly, or use outer class name.

6

Other Features of Inner Classes

- Outside of the outer class, use `outer.inner` notation to refer to type of inner class
 - E.g., `Queue.Entry`
- An inner class marked *static* does not have a reference to outer class
 - Can’t refer to instance variables of outer class
 - Must also use `outer.inner` notation to refer to inner class
- Question: Can `Queue.Element` be made static?

7

Anonymous Inner Classes

```
(new Thread() {
    public void run() {
        try {
            Thread.sleep(1000*60*20);
            System.out.println("...");
            System.exit(1);
        } catch (Exception e) {}
    }
}).start();
```

- Create anonymous subclass of thread, and invoke method on it

8

Compiling Inner Classes

- The JVM doesn't know about inner classes
 - Compiled away, similar to generics
 - Inner class Foo of outer class A produces A\$Foo.class
 - Anonymous inner class of outer class A produces A\$1.class
- Why are inner classes useful?

9

Iteration

- Goal: Loop through all objects in an aggregate

```
class Node { Element elt; Node next; }  
Node n = ...;  
while (n != null) { ...; n = n.next; }
```

- Problems:
 - Depends on implementation details
 - Varies from one aggregate to another

10

Iterators in Java

```
public interface Iterator {  
    // returns true if the iteration has more elts  
    public boolean hasNext();  
  
    // returns the next element in the iteration  
    public Object next() throws NoSuchElementException;  
}
```

(plus optional remove method)

- Implementation of aggregate not exposed
- Generic for wide variety of aggregates
- Supports multiple traversal strategies

11

Generic Iterators in Java 1.5

```
public interface Iterator<A> {  
    // returns true if the iteration has more elts  
    public boolean hasNext();  
  
    // returns the next element in the iteration  
    public A next() throws NoSuchElementException;  
}
```

12

Using Iterators

```
Iterator i = c.iterator();
while (i.hasNext()) {
    Element e = (Element) i.next();
    // do stuff with e
}

// alternatively use for
for (Iterator i = c.iterator(); i.hasNext(); ) {
    Element e = (Element) i.next();
    // do stuff with e
}
```

13

Iterators and Queues

- Recall queue example from beginning of lecture
- We'll explore options for adding iterators

14

next() Shouldn't Mutate Aggregate

```
class Queue<Element> {
    ...
    class QueueIterator implements Iterator<Element> {
        Entry rest;

        QueueIterator(Entry q) { rest = q; }
        boolean hasNext() { return rest != null; }
        Element next() throws NoSuchElementException {
            if (rest == null)
                throw new NoSuchElementException();
            Element e = rest.elt;
            rest = rest.next; // queue data intact
            return e;
        }
    }
}
```

15

Evil Mutating Clients

- But a client could mutate the data structure ...

```
HashMap h = ...;
...
Iterator i = h.entrySet().iterator();
System.out.println(i.next());
System.out.println(i.next());
h.put("Foo", "Bar"); // hash table resize!
System.out.println(i.next()); // prints ???
```

16

Defensive (Proactive) Copying

- Solution 1: Iterator copies data structure

```
class QueueIterator implements Iterator<Element> {
    Entry rest;

    QueueIterator(Queue q) {
        // copy q.theQueue to rest
    }
}
```

- Pro: Works even if queue is mutated
- Con: Expensive to construct iterator

17

Timestamps

- Solution 2: Track Mutations

```
class Queue<Element> {
    ...
    int modCount = 0;
    void enqueue(Element e) { ... modCount++; }
    Element dequeue() { ... modCount++; }

    ...
}
```

18

Timestamps (cont'd)

```
...
class QueueIterator implements Iterator<Element> {
    int expectedModCount = modCount; // set at iterator
                                        // construction time

    Element next() {
        if (expectedModCount != modCount)
            throw new ConcurrentModificationException();
        ...
    }
    // does hasNext() need to be modified?
}}
```

- Pro: Iteration construction cheap
- Con: Doesn't allow any mutation

19

Comments

- Neither solution tracks mutations to container elts
 - Could use clone(), but tricky

20

What if Mutation is Allowed?

- Allowed mutation must be part of iterator spec

```
public void remove()
    throws IllegalStateException;
```
- Removes from the underlying collection the last element returned by the iterator (optional operation). This method can be called only once per call to next.
- The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.

21

Iterators

- Key ideas
 - Separate aggregate structure from traversal protocol
 - Support additional kinds of traversals
 - E.g., smallest to largest, largest to smallest, unordered
 - Multiple simultaneous traversals
 - Though many Java Collections do not provide this
- Structure
 - Iterator interface defines traversal protocol
 - Concrete Iterator implementations for each aggregate
 - And for each traversal strategy
 - Aggregate instances create Iterator object instances

22

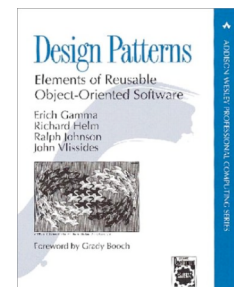
Design Patterns

- Iterators are an example of a *design pattern*:
 - Design pattern = problem + solution in context
 - Iterators: solution for providing generic traversals
- Design patterns capture software architectures and designs
 - Not direct code reuse!
 - Instead, solution/strategy reuse
 - Sometimes, interface reuse

23

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
- On reserve in CS library (3rd floor AVW)



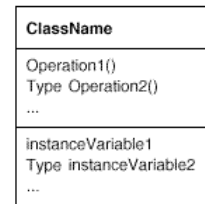
24

Object Modeling Technique (OMT)

- Used to describe patterns in GO4 book
- Graphical representation of OO relationships
 - **Class diagrams** show the static relationship between classes
 - **Object diagrams** represent the state of a program as series of related objects
 - **Interaction diagrams** illustrate execution of the program as an interaction among related objects

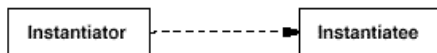
25

Classes



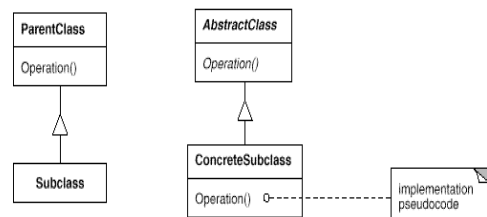
26

Object instantiation



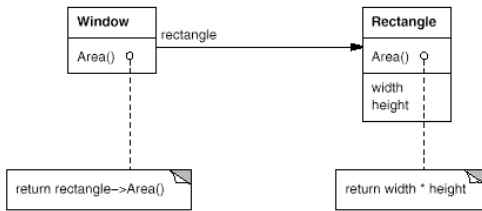
27

Subclassing and Abstract Classes



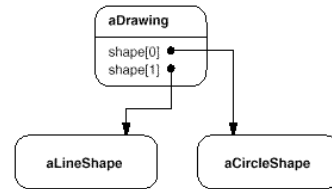
28

Pseudo-code and Containment



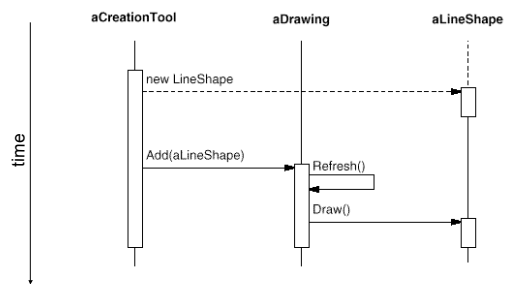
29

Object diagrams



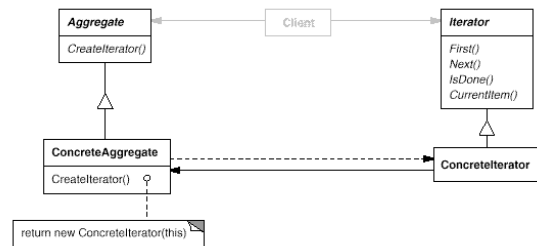
30

Interaction diagrams



31

Structure of Iterator (Cursor) Pattern



32

Components of a Pattern

- Name(s)
- Problem
 - Context
 - Real-world example
- Solution
 - Design/structure
 - Implementation
- Consequences
- Variations, known uses

33

Iterator Pattern, Again

- **Name:** Iterator (*aka* Cursor)
- **Problem:**
 - How to process the elements of an aggregate in an implementation-independent manner?
- **Solution:**
 - Define an Iterator interface
 - `next()`, `hasNext()`, etc. methods
 - Aggregate returns an instance of an implementation of Iterator interface to control the iteration

34

Iterator Pattern

- **Consequences:**
 - Support different and simultaneous traversals
 - Multiple implementations of Iterator interface
 - One traversal per Iterator instance
 - Requires coherent policy on aggregate updates
 - Invalidate Iterator by throwing an exception, or
 - Iterator only considers elements present at the time of its creation
- **Variations:**
 - Internal vs. external iteration
 - Java Iterator is external

35

Internal Iterators

```
public interface InternalIterator {  
    void iterate(Processor p);  
}  
  
public interface Processor {  
    public void process(Element e);  
}
```

- The internal iterator applies the processor instance to each element of the aggregate
 - Thus, entire traversal happens “at once”
 - Less control for client, but easier to formulate traversal

36

Example

```
iterator i = q.extIterate();
int count = 0;

while (i.hasNext()) {
    i.next();
    count++;
}
System.out.println(p.count);
```

External

```
class P implements Processor {
    int count = 0;
    public void process(Element e)
    { count++; }
}
```

```
Processor p = new Processor();
q.intIterate(p);
System.out.println(p.count);
```

Internal

37

CMSC 433 – Programming Language Technologies and Paradigms Spring 2005

Design Patterns
February 17, 2005

38

Design Patterns: Goals

- To support **reuse** of successful designs
- To facilitate **software evolution**
 - Add new features easily, without breaking existing ones
- In short, we want to **design for change**

39

Underlying Principles

- Reduce implementation dependencies between elements of a software system
- Sub-goals:
 - Program to an interface, not an implementation
 - Favor composition over inheritance
 - Use delegation

40

Program to Interface, Not Implementation

- Rely on abstract classes and interfaces to hide differences between subclasses from clients
 - Interface defines an object's use (protocol)
 - Implementation defines particular policy
- *Example: Iterator* interface, compared to its implementation for a **LinkedList**

41

Rationale

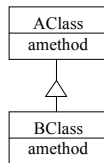
- Decouples clients from the implementations of the applications they use
- When clients manipulate an interface, they remain unaware of the specific object types being used.
- Therefore: clients are less dependent on an implementation, so it can be easily changed later.

42

Favor Composition over Class Inheritance

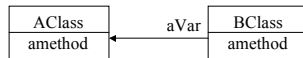
- White box reuse:

– Inheritance



- Black box reuse:

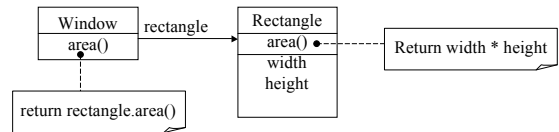
– Composition



43

Delegation

- Forward messages (delegate) to different instances at run-time; a form of composition
 - May pass invoking object's **this** pointer to simulate inheritance



44

Rationale

- White-box reuse has results in implementation dependencies on the parent class
 - Reusing a subclass may require rewriting the parent
 - But inheritance easy to specify
- Black-box reuse often preferred
 - Eliminates implementation dependencies, hides information, object relationships non-static for better run-time flexibility
 - But adds run-time overhead (additional instance allocation, communication by dynamic dispatch)
 - Sometimes code harder to read and understand

45

Design Patterns Taxonomy

- Creational patterns
 - Concern the process of object creation
- Structural patterns
 - Deal with the composition of classes or objects
- Behavioral patterns
 - Characterize the ways in which classes or objects interact and distribute responsibility

46

Catalogue of Patterns: Creation Patterns

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

47

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
- Proxy
 - Provide a surrogate or placeholder for another object to control access to it
- Decorator
 - Attach additional responsibilities to an object dynamically

48

Behavioral Patterns

- **Template**
 - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses
- **State**
 - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class
- **Observer**
 - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

49

Singleton Objects

- **Problem:**
 - Some classes have conceptually one instance
 - Many printers, but only one print spooler
 - One file system
 - One window manager
 - Creating many objects that represent the same conceptual instance adds complexity and overhead
- **Solution: only create one object and reuse it**
 - Encapsulate the code that manages the reuse

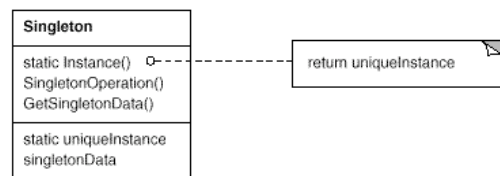
50

The Singleton Solution

- **Class is responsible for tracking its sole instance**
 - Make constructor private
 - Provide static method/field to allow access to the only instance of the class
- **Benefit:**
 - Reuse implies better performance
 - Class encapsulates code to ensure reuse of the object; no need to burden client

51

Singleton pattern



52

Implementing the Singleton method

- In Java, just define a final static field

```
public class Singleton {  
    private Singleton() {...}  
  
    final private static Singleton instance  
        = new Singleton();  
  
    public static Singleton getInstance()  
    { return instance; }  
}
```

- Java semantics guarantee object is created immediately before first use

53

Marshalling

- *Marshalling* is the process of transforming internal data into a form that can be
 - Written to disk
 - Sent over the network
 - Etc.
- *Unmarshalling* is the inverse process

54

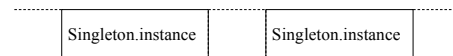
Marshalling in Java

- Java provides support for marshalling objects
 - Classes implement the *Serializable* interface
 - The JVM implements standard marshalling and unmarshalling automatically
 - E.g., enables you to create persistent objects, stored on disk
 - This can be useful for build a light-weight database
 - Also useful for distributed object systems
- Often, generic implementation works fine
 - But let's consider singletons...

55

Marshalling and Singletons

- What happens when we unmarshall a singleton?



- Problem: JVM doesn't know about singletons
 - It will create two instances of Singleton.instance!
 - Oops!

56

Marshalling and Singletons (cont'd)

- Solution: Implement
 - Object readResolve() throws ObjectStreamException;
 - This method will be called after standard unmarshalling
 - Returned result is substituted for standard unmarshalled result
- E.g., add to Singleton class the following method
 - Object readResolve() { return instance; }
- Do we need to worry about marshalling?
- Notes: Serialization is a big hack!
 - Doesn't follow language conventions

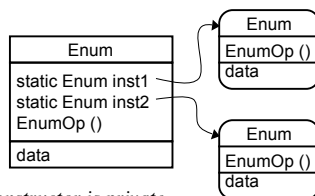
57

Generalizing Singleton: Typesafe Enum

- Problem:
 - Need a number of unique objects, not just one
 - Basically want a C-style enumerated type, but safe
- Solution:
 - Generalize the Singleton Pattern to keep track of multiple, unique objects (rather than just one)

58

Typesafe Enum Pattern



Note: constructor is private

59

Typesafe Enum: Example

```
public class Suit {
    private final String name;

    private Suit(String name) { this.name = name; }

    public String toString() { return name; }

    public static final Suit CLUBS = new Suit("clubs");
    public static final Suit DIAMONDS = new Suit("diamonds");
    public static final Suit HEARTS = new Suit("hearts");
    public static final Suit SPADES = new Suit("spades");
}
```

- Exercise: What about serialization?

60

Enumerators in Java 1.5

- New version of Java has type safe enums
 - Built-in: Don't need to use the design pattern
- ```
public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
```

  - Type checked at compile time
  - Implemented as objects (translated as prev slide?)
  - Two extra class methods:
    - `public static <this enum class>[] values()` -- the enumeration elts
    - `public static <this enum class> valueOf(String name)` -- get an elt

61

## Adapter (aka Wrapper) Pattern

- Problem:
  - You have some code you want to use for a program
  - You can't incorporate the code directly (e.g., you just have the .class file, say as part of a library)
  - The code does not have the interface you want
    - Different method names
    - More or fewer methods than you need
- To use this code, you must *adapt* it to your situation

62

## Adapter Pattern (cont'd)

- Here's what we have:



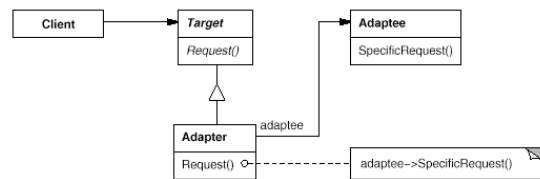
- Client is already written, and it uses the Target interface
- Adaptee has a method that works, but has the wrong name

- How do we enable the Client to use the Adaptee?

63

## Adapter Pattern (cont'd)

- Solution: adapter class to implement client's expected interface, forwarding methods



64

## Proxy Pattern Motivation

- Goal:
  - Prevent an object from being accessed directly by its clients
- Solution:
  - Use an additional object, called a proxy
  - Clients access protected object only through proxy
  - Proxy keeps track of status and/or location of protected object

65

## Uses of the Proxy Pattern

- *Virtual proxy*: impose a lazy creation semantics, to avoid expensive object creations when strictly unnecessary.
- *Monitor proxy*: impose security constraints on the original object, say by intercepting some method calls to proxied object.
- *Remote proxy*: hide the fact that an object resides on a remote location.

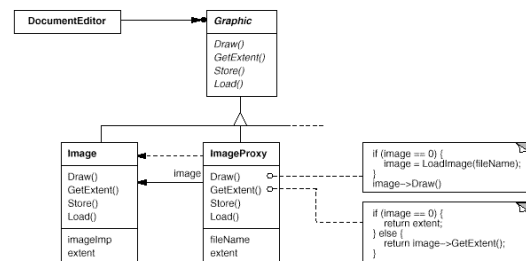
66

## More OMT Notation

- Arrow ending in filled circle
  - More than one

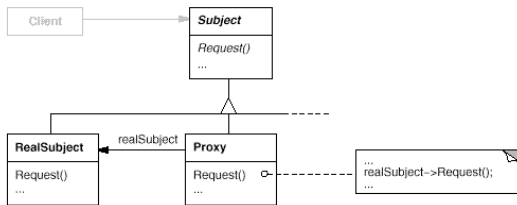
67

## Example Usage Class Diagram



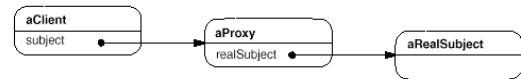
68

## Proxy Pattern Class Diagram



69

## Object Diagram



70

## Proxy vs. Adapter

- Proxies implement *the same* interface as the objects they adapt
  - But may restrict some operations
  - E.g., refuse to perform a sensitive operation
- Adapters implement *a different* interface than the objects they adapt

71

## Decorator Pattern

- Motivation
  - Want to add responsibilities/capabilities to individual objects, not to an entire class
  - Inheritance requires a compile-time choice of parent class
- Solution
  - Enclose the component in another object that adds the responsibility/capability
    - The enclosing object is called a **decorator**.

72

## Example: Java I/O

```

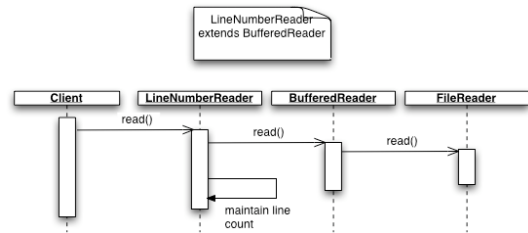
class LineNumberReader extends BufferedReader {
 private int lineNumber;
 public LineNumberReader(Reader in) { super(in); }
 public int getLineNumber() { return lineNumber; }

 public int read() {
 int c = super.read();
 if (c == '\n') {
 lineNumber++;
 return '\n';
 }
 return c;
 }
}

```

73

## Interaction Diagram



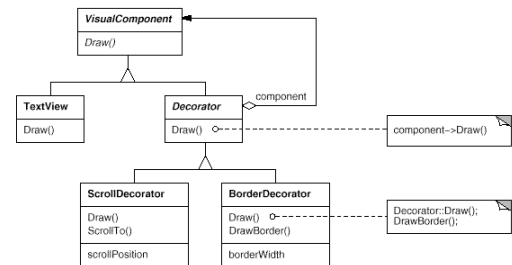
74

## More OMT Notation

- Arrow beginning with diamond
  - "Part-of" or aggregation
  - Only accessed by object pointing to it

75

## Decorator Pattern: Another Example



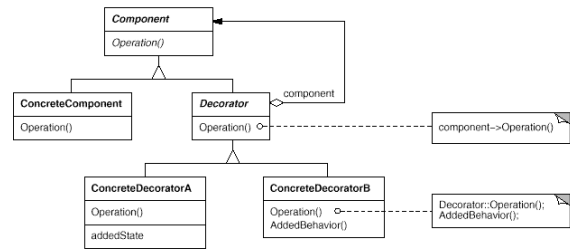
76

## Decorator Pattern: Features

- Decorator conforms to interface of decorated component
  - Its presence is transparent to the component's clients.
- Decorator forwards requests to encapsulated component
  - May perform additional actions before or after
- Can nest decorators recursively
  - Allows unlimited added responsibilities
- Can add/remove responsibilities dynamically

77

## Structure



78

## Decorator Pattern Analysis

- Advantages
  - Fewer classes than with static inheritance
  - Dynamic addition/removal of decorators
  - Keeps root classes simple
- Disadvantages
  - Proliferation of run-time instances
  - Abstract Decorator must provide common interface
- Tradeoffs:
  - Useful when components are lightweight
  - Otherwise use Strategy

79

## Decorator vs. Adapter

- A decorator *adds* to the responsibilities of an object
  - Usually has the same interface plus more features
- An adapter *changes* the interface
  - But usually not the responsibilities

80

# CMSC 433 – Programming Language Technologies and Paradigms Spring 2005

Design Patterns  
February 22, 2005

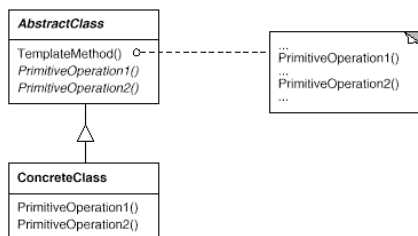
81

## Template Method Pattern

- Problem
  - You're building a reusable class
  - You have a general approach to solving a problem
  - But each subclass will do things differently
- Solution
  - Invariant parts of an algorithm in parent class
  - Encapsulate variant parts in template methods
  - Subclasses override template methods
  - At runtime template method invokes subclass ops

82

## Structure



83

## Example from JDK

```
public abstract class OutputStream {
 public abstract void write(int b) ...;
 public void write(byte b[], int off, int len) ... {
 ... write(b[off + i]);...
 }
 ...
}
```

- Subclasses of `OutputStream` need not override the second version of `write(...)`
  - But they do need to override the first one, since it's abstract
  - (Note: They may want to override anyhow for efficiency)

84

## Example from Project 1

```
public abstract class ServletFilter extends
 OutputStream implements MiniServlet
{
 public abstract void write(int b) ...;
 public abstract void close() ...;
 public abstract void flush() ...;

 public void setArg(String arg);
 public void setOutputStream(OutputStream out);
}
```

85

## Example: JUnit

- Junit uses template methods pattern for **run()**

```
package junit.framework;
public class TestCase {
 ...
 public void run() {
 setUp(); runTest(); tearDown()
 }
}
```

- In class example, we subclass TestCase and override setUp() and tearDown()

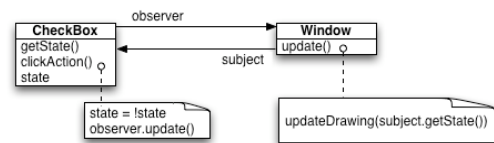
86

## Observer Pattern

- Problem
  - One object must be consistent with another's state
- Solution
  - When subject object change state, notify the observing object

87

## Example: GUI



- Window takes some action whenever checkbox is changed

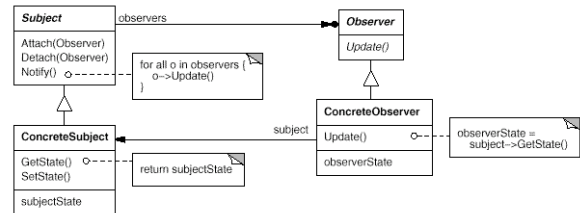
88

## Solution Structure

- Solution structure: Four kinds of objects
  - Abstract subject
    - Maintain list of dependents; notifies them when master changes
  - Abstract observer
    - Define protocol for updating dependents
  - Concrete subject
    - Manage data for dependents; notifies them when master changes
  - Concrete observers
    - Get new subject state upon receiving update message

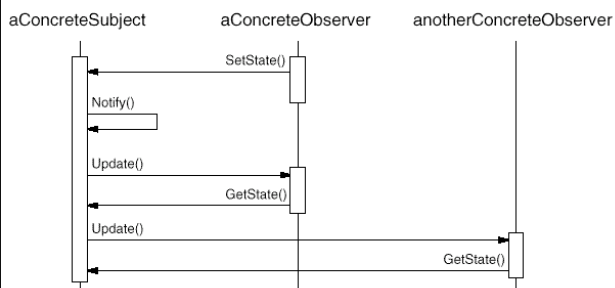
89

## Observer Pattern



90

## Use of Observer Pattern



91

## Observer Pattern (cont'd)

- Consequences
  - Low coupling between subject and observers
    - Subject unaware of dependents
  - Support for broadcasting
    - Dynamic addition and removal of observers
  - Unexpected updates
    - No control by the subject on computations by observers

92

## Observer Pattern (cont'd)

- Implementation issues
  - Storing list of observers
    - Typically in subject
  - Observing multiple subjects
    - Typically add parameters to update()
  - Who triggers update?
    - State-setting operations of subject
      - Possibly too many updates
    - Client
      - Error-prone if an observer forgets to send notification message

93

## Observer Pattern (cont'd)

- Implementation issues (cont'd)
  - Possibility of dangling references when subject is deleted
    - Easier in garbage-collected languages
    - Subject notifies observers before dying
  - How much information should subject send with update() messages?
    - Push model: Subject sends all information that observers may require
      - May couple subject with observers (by forcing a given observer interface)
    - Pull model: Subject sends no information
      - Can be inefficient

94

## Observer Pattern (cont'd)

- Implementation issues (cont'd)
  - Registering observers for certain events only
    - Use notion of an aspect in subject
    - Observer registers for one or more aspects
  - Complex updates
    - Use change managers
    - Change manager keeps track of complex relations among (possibly) many subjects and their observers and encapsulates complex updates to observers

95

## State Pattern

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

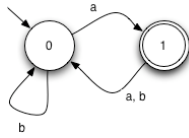
96

## Example – FSM, without State Pattern

```
class FSM {
 int state = 0; // or use enum

 public void move(char c) {
 if (state == 0) {
 if (c == 'a') state = 1;
 else if (c == 'b') state = 0;
 }
 if (state == 1) {
 if (c == 'a') state = 0;
 else if (c == 'b') state = 1;
 }
 }

 public boolean accept() {
 if (state == 0)
 return false;
 else if (state == 1)
 return true;
 }
}
```



97

## State Pattern Approach

- Encode different states as objects with same superclass
- To change state, change the state object
- Methods delegate to state object

98

## Example – with State Pattern

```
class FSM {
 State state;
 public FSM(State s) { state = s; }
 public void move(char c) { state = state.move(c); }
 public boolean accept() { return state.accept(); }
}

public interface State {
 State move(char c);
 boolean accept();
}
```

99

## FSM Example – cont.

```
class State0 implements State {
 static State0 instance = new State0();
 private State0() {}

 public State move(char c) {
 if (c == 'a')
 return State1.instance;
 else if (c == 'b')
 return State0.instance;
 }

 public boolean accept() {
 return false;
 }
}

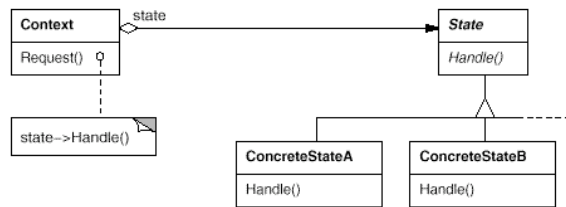
class State1 implements State {
 static State1 instance = new State1();
 private State1() {}

 public State move(char c) {
 if (c == 'a')
 return State0.instance;
 else if (c == 'b')
 return State0.instance;
 }

 public boolean accept() {
 return true;
 }
}
```

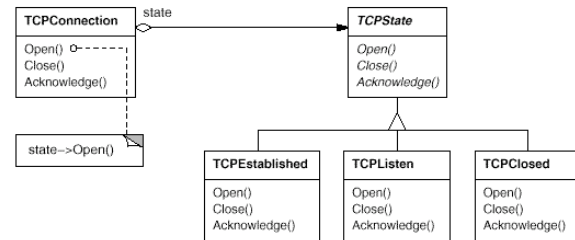
100

## Structure of State Pattern



101

## Instance of State Pattern



102

## State Pattern Notes

- Can use singletons for instances of each state class
  - State objects don't encapsulate (mutable) state, so can be shared
- Easy to add new states
  - New states can extend the base class, or
  - New states can extend other states
    - Override only selected functions

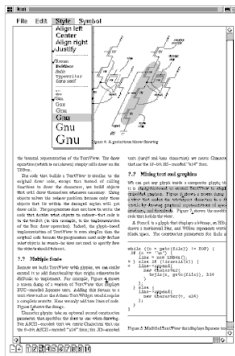
103

## Lexi: Simple GUI-Based Editor

- Lexi is a WYSIWYG editor
  - Supports documents with textual and graphical objects
  - Scroll bars to select portions of the document
  - Be easy to port to another platform
  - Support multiple look-and-feel interfaces
- Highlights several OO design issues
- Case study of design patterns in the design of Lexi

104

## Lexi User Interface



105

## Design Issues

- Representation and manipulation of document
- Formatting a document
- Adding scroll bars and borders to Lexi windows
- Support multiple look-and-feel standards
  - Motif and Presentation Manager (!)
- Handle multiple windowing systems
- Support user operations
- Advanced features
  - spell-checking and hyphenation

106

## Structure of a Lexi Document

- Goals:
  - Store text and graphics in document
  - Generate visual display
  - Maintain info about location of display elements
- Caveats:
  - Treat different objects uniformly
    - E.g., text, pictures, graphics
  - Treat individual objects and groups of objects uniformly
    - E.g., characters and lines of text

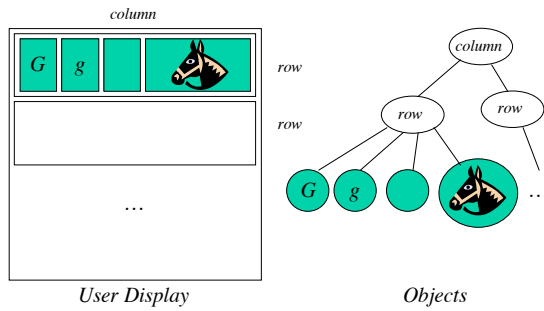
107

## Structure of a Lexi Document

- Use **recursive composition** for defining and handling complex objects
  - Abstract class **Glyph** for all displayed objects
  - Glyph responsibilities:
    - Know how to draw itself
    - Knows what space it occupies
    - Knows its children and parent
  - Glyph instances can recursively **compose** other Glyph instances

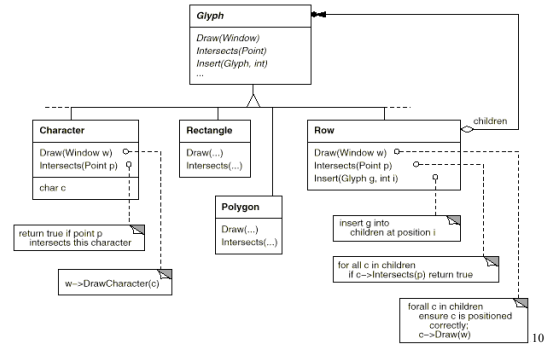
108

## Recursive Composition



109

## Glyph Class Diagram



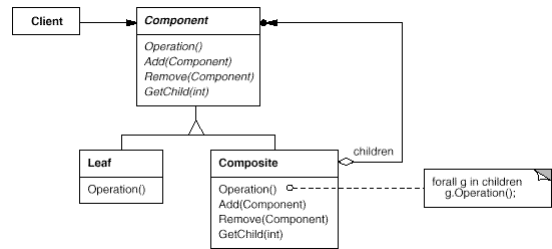
10

## The Composite Pattern

- Motivation:
  - Support recursive composition in such a way that a client need not know the difference between a single and a composite object (as with Glyphs)
- Applicability:
  - When dealing with hierarchically-organized objects (e.g., columns containing rows containing words ...)

111

## Composite Pattern Structure



112

## Composite Pattern Consequences

- Class hierarchy has both **simple** and **composite** objects
- Simplifies clients
- Aids extensibility
  - Clients do not have to be modified
- Too general a pattern?
  - Difficult to restrict functionality of concrete leaf subclasses

113

## Formatting Lexi Documents: Strategy

- We know that documents are represented as Glyphs, but not how documents are constructed.
- Formatting:
  - Document structure will be determined based on rules for justification, margins, line breaking, etc.
  - Many good algorithms exist;
    - different tradeoffs between quality and speed
- Design decision: implement different algorithms, decide at run-time which algorithm to use
  - define root class that supports many algorithms
  - each algorithm implemented in a subclass

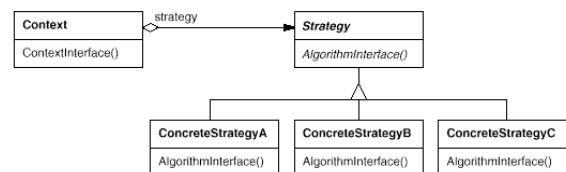
114

## Strategy Pattern

- Name
  - Strategy (aka Policy)
- Applicability
  - Many related classes differ only in their behavior
  - Many different variants of an algorithm
  - Need to encapsulate algorithmic information

115

## Strategy Pattern: Structure



116

## Strategy Pattern: Consequences

- Clear separation of algorithm definition and use
  - Glyphs and formatting algorithms are independent
  - Alternative (many subclasses) is unappealing
    - Proliferation of classes
    - Algorithms cannot be changed dynamically
- Elimination of conditional statements
  - Like State, Template, ...
  - Typical in OO programming

117

## Strategy Pattern Consequences (cont'd)

- Clients must be aware of different strategies
  - When initializing objects
- Proliferation of instances at run-time
  - Each Glyph has a strategy object with formatting information
  - If strategy is stateless, share strategy objects

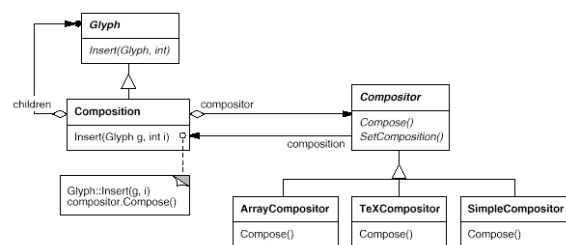
118

## Lexi: Using Strategy

- Compositor and Composition classes
  - Compositor: class encapsulating formatting algorithm
    - Pass Composition objects to be formatted as parameters to Compositor methods
  - Composition: things being formatted
    - Glyph subclass
    - Each Composition object refers to its Compositor object
    - When a Composition needs to format itself, it sends a message to its Compositor instance

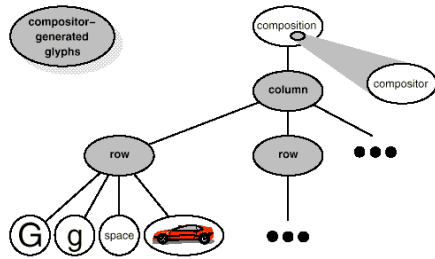
119

## Class Diagram



120

## Object Structure after Formatting



121

## CMSC 433 – Programming Language Technologies and Paradigms Spring 2005

Design Patterns  
March 1, 2005

122

## Spell-Checking and Hyphenation

- Must do textual analysis
  - Multiple operations and implementations
- Must add new functions and operations easily
- Must efficiently handle scattered information and varied implementations
  - Different traversal strategies for stored information
- Should separate actions from traversal

123

## Visitor: Implementing Analyses

- Often want to implement multiple analyses on the same kind of object data
  - Spellchecking and Hyphenating Glyphs
  - Generating code for and analyzing an Abstract Syntax Tree (AST) in a compiler
- One solution: implement each analysis as a method in each object

124

## Abstract Syntax Trees

```
public interface Node { }

public class Number extends Node {
 public int n;
}

public class Plus extends Node {
 public Node left;
 public Node right;
}
```

125

## Traversing Abstract Syntax Trees

```
public interface Node {
 public int sum();
}

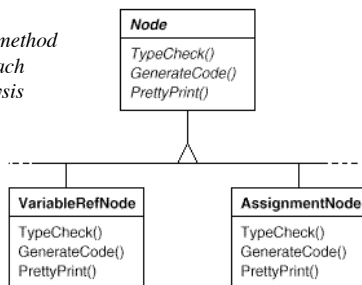
public class Number extends Node {
 public int n;
 public int sum() { return n; }
}

public class Plus {
 public Node left;
 public Node right;
 public int sum() { return left.sum() +
 right.sum(); } } }
```

126

## Naïve approach (not a visitor)

*One method  
for each  
analysis*



127

## Tradeoffs with this Approach

- Follows idea “objects are responsible for themselves”
- But many analyses will occlude the object’s main code
- Result is classes that are hard to maintain

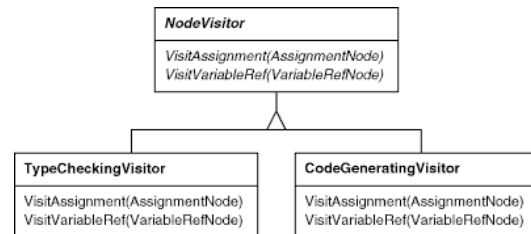
128

## Use a Visitor

- Alternatively, can define a separate **visitor** class
  - A visitor encapsulates the operations to be performed on an entire structure, e.g., all elements of a parse tree
- Allows operations to be separate from structure
  - But doesn't necessarily require putting all of the structure traversal code into each visitor/operation

129

## Sample Visitor class



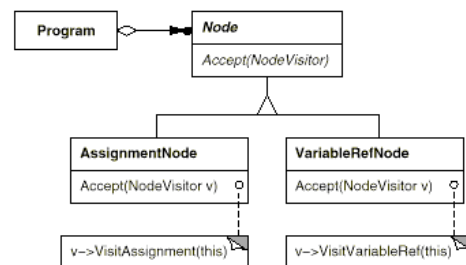
130

## How to perform traversal?

- Now that we have a visitor class, how do we apply its analysis to the objects of interest?
  - Add **accept(visitor)** method to each structure class, that will invoke the given visitor on **this**
  - Builds on Java's dynamic dispatch
  - Use an iteration algorithm (like an Iterator) to call **accept()** on each relevant object

131

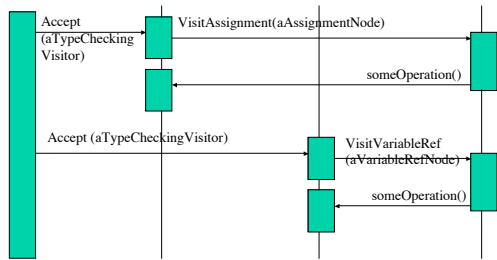
## Sample visited objects



132

## Visitor Interaction

*aNodeStructure aAssignmentNode aVariableRefNode aTypeCheckingVisitor*



133

## Sample Visitor Class

```

public interface Visitor {
 public void visitNumber(Number n);
 public void visitPlus(Plus p);
}

public class SumVisitor implements Visitor {
 int sum;
 public void visitNumber(Number n) { sum += n; }
 public void visitPlus(Plus p) {
 p.left.accept(this);
 p.right.accept(this);
 }
}

```

134

## Change to AST Classes

```

public interface Node {
 public void accept(Visitor v);
}

public class Number extends Node {
 ...
 public void accept(Visitor v) {v.visitNumber(this);}
}

public class Plus extends Node {
 ...
 public void accept(Visitor v) {v.visitPlus(this);}
}

```

135

## Visitor pattern

- Name
  - Visitor or double dispatching
- Applicability
  - Related objects must support different operations and actual op depends on both the class and the op type
  - Distinct and unrelated operations pollute class defs
  - **Key:** object structure rarely changes, but ops changed often

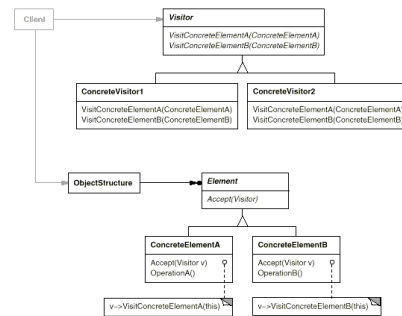
136

## Visitor Pattern Structure

- Define two class hierarchies
  - One for object structure
    - AST in compiler, Glyphs in Lexi
  - One for each operation family, called visitors
    - One for typechecking, code generation, pretty printing in compiler
    - One for spellchecking or hyphenation in Lexi

137

## Structure of Visitor Pattern



138

## Visitor Pattern Consequences

- Adding new operations is easy
  - Add new op subclass with method for each concrete elt class
  - Easier than modifying every element class
- Gathers related operations and separates unrelated ones
- Adding new concrete elements is difficult
  - Must add a new method to each concrete Visitor subclass
- Allows visiting across class hierarchies
  - Iterator needs a common superclass (i.e., composite pattern)
- Visitor can accumulate state rather than pass it as parameters

139

## 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*
  - Traversal routine takes two arguments, the visitor and the object to traverse
    - `o.accept(aVisitor)` will dispatch on the actual identity of `o` (the object being considered)
    - ...and `accept` will internally dispatch on the identity of `aVisitor` (the object visiting it)

140

## 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 by compile-time overload resolution

141

## Visitors Can Forward Common Behavior

- Useful for composites
  - If subclasses of a particular object all treated the same
  - Can have visit(SubClass) call visit(SuperClass)
- For example
  - visit(BinaryPlusOperatorNode) can just forward call to superclass visit(BinaryOperatorNode)

142

## State in a Visitor Pattern

- A visitor can contain state
    - E.g., the results of typechecking the program so far
- ```
class TypeCheckingVisitor extends Visitor {  
    private TypeMap map;  
    void visit(VariableDefNode n) { ...  
        map.add(n, t)  
    ... }  
}
```
- Or visitors pass around a separate state object
 - Impacts the type of the Visitor superclass

143

Implementing Traversal

- Who is responsible for traversing object structure?
- Plausible answers:
 - Visitor
 - But, must replicate traversal code in each concrete visitor
 - Object structure
 - Define operation that performs traversal while applying visitor object to each component
 - Iterator
 - Iterator sends message to visitor with current element as arg

144

Traversals

- It's sometimes preferable to try to keep traversal separate from the Visitor
 - E.g., use an Iterator
 - Thus traversal and analysis can evolve independently
- But can also do it within node or visitor class. Several solutions here:
 - **acceptAndTraverse** methods
 - traverse from within accept()
 - Separating processing from traversal
 - Visit/process methods
 - Traversal visitors applying an operational visitor

145

Accept and Traverse Example

- Class BinaryPlusOperatorNode {

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

146

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

147

Visitor/Process Methods

- Can have two parallel sets of methods in visitors
 - Visit() methods
 - Process() methods
- How it works: the 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)
- Allows finer-grained subtyping of Visitor classes that include traversal
 - Subclass a visitor, and just change the process method₁₄₈

Preorder Visitor

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

149

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

150

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

151

PreorderVisitor with Payload

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

152

CMSC 433 – Programming Language Technologies and Paradigms Spring 2005

Design Patterns
March 3, 2005

153

Changing Look-and-Feel: Abstract Factory

- Goal: easily change Lexi's look-and-feel
 - When new libraries are available (future variability)
 - At run-time by switching between them (present variability)
- Thoughtless implementation technique:
 - Use distinct class for each widget and standard
 - Let clients handle different instances for each standard
 - `Button pb = new MotifButton(); // bad`

154

Abstracting Creation

- Concrete creation problems:
 - Class of object is fixed at compile-time
 - Can't change standard at run-time
 - Changing the class means making changes all over the code
- Instead:
 - Use a class to create abstract classes:
 - `Button pb = guiFactory.createButton(); // better`

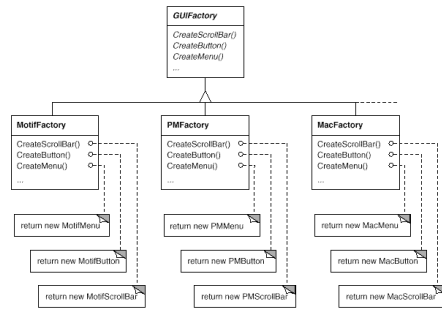
155

Solution: Use Abstract Factory

- Define abstract class GUIFactory with creation methods for widgets
 - Concrete subclasses of GUIFactory actually define creation methods for each look-and-feel standard
 - `MotifFactory, MacFactory, etc.`
 - Specialize each widget into subclasses for each look-and-feel standard
- Thus, can easily change the kind of factory without changes all over the place

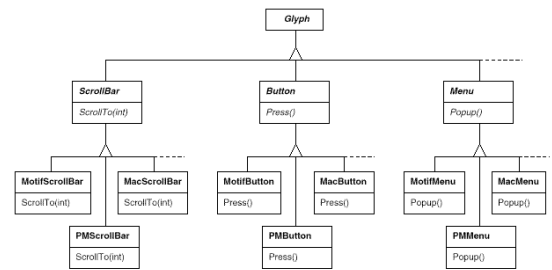
156

Class Diagram for GUIFactory



157

Diagram for Product Classes



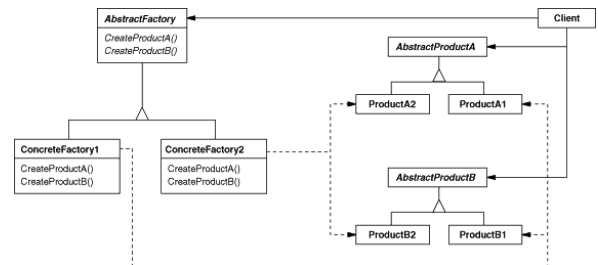
158

Abstract Factory Pattern

- Name
 - Abstract Factory or Kit
- Applicability
 - Different families of components (products)
 - Must be used in mutually exclusive and consistent way
 - Hide existence of multiple families from clients

159

Structure of Abstract Factory



160

Abstract Factory: Consequences

- Isolate instance creation and handling from clients
- Can easily change look-and-feel standard
 - Reassign a global variable
 - Recompute and redisplay the interface
- Enforce consistency among products in each family
- Adding to family of products is difficult
 - Have to update factory abstract class and all concrete classes

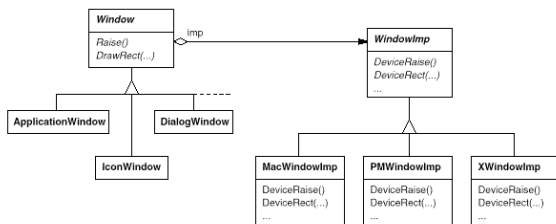
161

Multiple Window Systems

- Want portability to different window systems
 - Similar to multiple look-and-feel problem, but different vendors will build widgets differently
- Solution:
 - Define abstract class Window, with basic window functionality (e.g., draw, iconify, move, resize, etc.)
 - Define concrete subclasses for specific types of windows (e.g., dialog, application, icon, etc.)
 - Define WindowImp hierarchy to handle window implementation by a vendor

162

Implementation



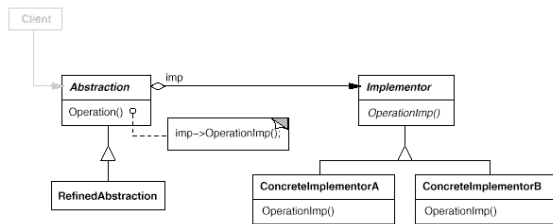
163

Bridge Pattern

- Name
 - Bridge or Handle or Body
- Applicability
 - Handles abstract concept with different implementations
 - Implementation may be switched at run-time
 - Implementation changes should not affect clients
 - Hide a class's interface from clients
- Structure: use two hierarchies
 - Logical one for clients
 - Physical one for different implementations

164

Structure of Bridge Pattern



165

Bridge Pattern

- Consequences:
 - Decouple interface from implementation and representation
 - Change implementation at run-time
 - Improve extensibility
 - Logical classes and physical classes change independently
 - Hides implementation details from clients
 - Sharing implementation objects and associated reference counts

166

Supporting User Commands

- Support execution of Lexi commands
 - GUI doesn't know
 - Who command is sent to
 - Command interface
- Complications
 - Different commands have different interfaces
 - Same command can be invoked in different ways
 - Undo and Redo for some, but not all, commands (print)

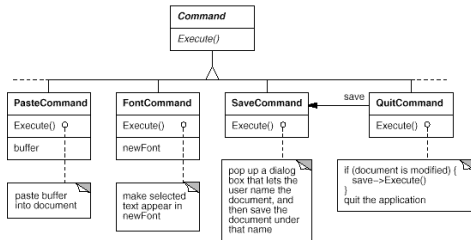
167

Supporting User Commands (cont'd)

- An improved solution
 - Create abstract "command" class
 - Create action-performing glyph subclass
 - Delegate action to command
- Key ideas
 - Pass an object, not a function
 - Pass context to the command function
 - Store command history

168

Command Objects



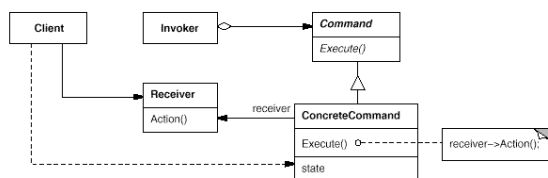
169

Command Pattern

- Name
 - Command or Action or Transaction
- Applicability
 - Parameterize objects by actions they perform
 - Specify, queue, and execute requests at different times
 - Support undo by storing context information
 - Support change log for recovery purposes
 - Support high-level operations
 - Macros

170

Structure of Command Pattern



171

Command Pattern

- Consequences:
 - Decouple receiver and executor of requests
 - Lexi example: Different icons can be associated with the same command
 - Commands are first class objects
 - Easy to support undo and redo
 - Command must have method to check whether it's reversible
 - Must add state information
 - Can create composite commands
 - Editor macros
 - Can extend commands more easily

172

Command Pattern

- Implementation notes
 - How much should command do itself?
 - Support undo and redo functionality
 - Operations must be reversible
 - May need to copy command objects
 - Don't record commands that don't change state
 - Avoid error accumulation in undo process

173

Comparing Objects

- Java has two designs for objects that can be (totally) ordered
 - These are things for which sorting makes sense
 - E.g., strings, integers, etc.

174

Comparable

```
public interface Comparable {  
    // Returns negative integer, zero, or a positive integer if this  
    // object is less than, equal to, or greater than o.  
    public int compareTo(Object o);  
}
```

- Advantages and disadvantages?
 - Can only implement one compareTo operation
 - No extra levels of indirection; objects know how to compare themselves

175

Comparator

```
public interface Comparator {  
    int compare(Object o1, Object o2);  
}
```

- Advantages and disadvantages?
 - Can have multiple comparison operations
 - An example of delegation
 - Comparator needs to know innards of your objects
 - Can make the Comparator implementer an inner class
 - Extra indirection; more objects floating around

176

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++ can have very different solutions in Smalltalk or Java