

## CMSC 433 – Programming Language Technologies and Paradigms Spring 2003

Iterators and Design Patterns  
February 27, 2003

1

## 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

2

## 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

3

## Iterators in Java

```
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;  
}
```

(plus optional remove method)

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

4

## Using Iterators

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

5

## Using Iterators with for

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

- i only in scope within body of for loop
- slightly more compact (initialization, test on one line)
- slightly more cryptic

6

## The Queue Class

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

  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);
    }
  }
  ...
}
```

7

## The Queue Class

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

8

## 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;
      return e;
    }
  }
}
```

9

## Evil Mutating Clients

- even if next() won't mutate the data structure  
– ...a client could

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

10

## Defensive 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

11

## Timestamps

- Solution 2: Track Mutations

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

12

## 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

13

## Comments

- Neither solution tracks mutations to container elts
  - Could use clone(), but tricky
- Rarely, mutation is not an issue
  - E.g., try writing an iterator for Stack<Element>

14

## 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.

15

## 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
- Structure
  - Abstract Iterator class defines traversal protocol
  - Concrete Iterator subclasses for each aggregate
    - And for each traversal strategy
  - Aggregate instances create Iterator object instances

16

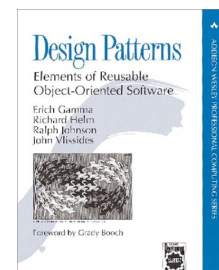
## Design Patterns

- 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 code reuse!
  - Instead, solution/strategy reuse
  - Sometimes, interface reuse

17

## 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 (3<sup>rd</sup> floor AVW)



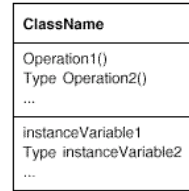
18

## 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

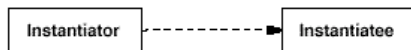
19

## Classes



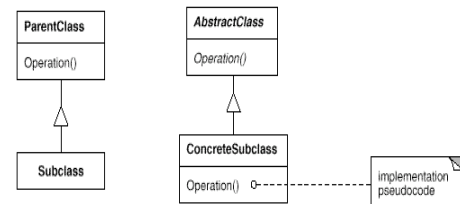
20

## Object instantiation



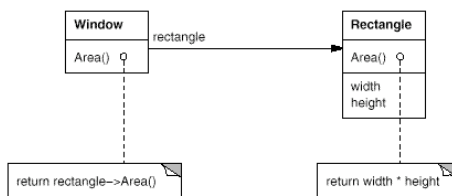
21

## Subclassing and Abstract Classes



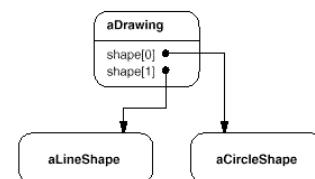
22

## Pseudo-code and Containment



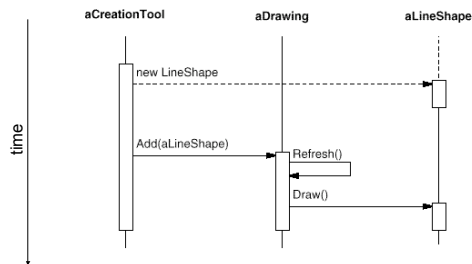
23

## Object diagrams



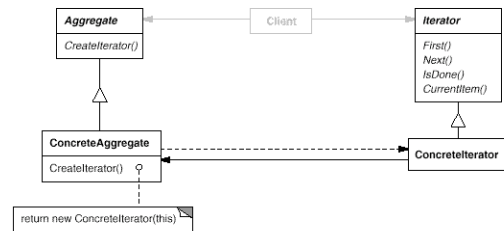
24

## Interaction diagrams



25

## Structure of Iterator (Cursor) Pattern



26

## Iterator Pattern (cont'd)

- Consequences
  - support different kinds of traversal strategies
    - just change Iterator instance
  - simplify aggregate's interface
    - no traversal protocols
  - supports simultaneous traversals

27

## Iterator Pattern (cont'd)

- Who controls the iterator?
- External vs. internal iterators
  - external:
    - client controls the iteration via “next” operation
    - very flexible
    - some operations are simplified - logical equality and set operations are difficult otherwise
  - internal:
    - Iterator applies operations to aggregate elements
    - easy to use
    - can be difficult to implement in some languages

28

## Iterator Pattern (cont'd)

- Who defines the traversal algorithm?
  - Iterator itself
    - may violate encapsulation
  - aggregate (in a “cursor”)
    - Iterator keeps only state of iteration
- How robust is the Iterator?
  - are updates or deletions handled?
  - don't want to copy aggregates
  - register Iterators with aggregate and clean-up as needed
  - synchronization of multiple Iterators is difficult

29

## Components of a Pattern

- Pattern name
  - identify this pattern; distinguish from other patterns
  - define terminology
- Pattern alias – “also known as”
- Real-world example
- Context
- Problem

30

## Components of a Pattern (cont'd)

- Solution
  - typically natural language notation
- Structure
  - class (and possibly object) diagram in solution
- Interaction diagram (optional)
- Consequences
  - advantages and disadvantages of pattern
  - ways to address residual design decisions

31

## Components of a Pattern (cont'd)

- Implementation
  - critical portion of plausible code for pattern
- Known uses
  - often systems that inspired pattern
- References - See also
  - related patterns that may be applied in similar cases

32

## 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.

33

## 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.

34

## 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

35

## 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

36

## Underlying Principles

- Program to an interface, not an implementation
- Favor composition over inheritance
- Use Delegation

37

## Program to Interface, Not Implementation

- Rely on abstract classes to hide differences between subclasses from clients
  - object class vs. object type
    - class defines how an object is implemented
    - type defines an object's interface (protocol)

38

## Favor Composition over Class Inheritance

- Black-box vs. white-box reuse
  - black-box relies on object references, usually through instance variables
  - white-box reuse by inheritance
  - black-box reuse preferred for information hiding, run-time flexibility, elimination of implementation dependencies
  - disadvantages: Run-time efficiency (high number of instances, and communication by message passing)

39

## Delegation

- Powerful technique when coupled with black-box reuse
- Allow delegation to different instances at run-time, as long as instances respond to similar messages
- Disadvantages:
  - sometimes code harder to read and understand
  - efficiency (because of black-box reuse)

40