Design Patterns

Iterator, Comparator, and nested classes

Dr. Chad Williams
Central Connecticut State University

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."

- Martin Golding

Agenda

- Design pattern: Iterator
- Nested classes – when to use them
- Comparator

Design guideline

- Program to an interface, not an implementation
  - Separate interface from implementation
  - Clients access functionalities via interface not directly on class
  - Implementation hidden from clients
- Programming to a class → context-specific, inflexible solutions
- Programming to an interface → general, extensible, reusable solutions

Enumerating elements

- Scenario – Given a group of Objects
- Operation - Loop through elements
- Common operation common to many different data structures

➢ Multiple implementations possible, used in multiple contexts making it a prime candidate for generalization

Solution 1 – direct access

```
LinkedList list;
for (LinkedList.Node cur = list.head;
        cur != null; cur = cur.next){
   System.out.println(cur.element);
}
```

- Solution works for LinkedList but tightly coupled to implementation breaking encapsulation
- Also would have to completely recode if a different data structure was wanted

Solution 2 – Iterate via Method invocation

```
public class IterList extends LinkedList{
  public void reset(){ cur = head; }
  public Object next(){
    Object obj = null;
    if (cur != null){
      obj = cur.element;
      cur = cur.next;
    }
    return obj;
  }
  public boolean hasNext(){
    return (cur != null);
  }
  protected Node cur;
}
```

- Encapsulates implementation but still strongly tied to context of implementation (ex. Looping within a loop)

Solution 3 – Separate iterator from List

```java
public class LinkedListIterator{
  public LinkedListIterator(LinkedList list){
    this.list = list;
    cur = list.head;
  }
  public Object next(){
    Object obj = null;
    if (cur != null){
      obj = cur.element;
      cur = cur.next;
    }
    return obj;
  }
  public boolean hasNext(){
    return (cur != null);
  }
  protected LinkedList.Node cur;
  protected LinkedList list;
}
```

- Fixes the multiple iterator problem but iterator is still tightly coupled with LinkedList

Solution 4: Generalization through abstract coupling

```
interface Iterator{
  Object next();
  boolean hasNext();
  void remove();
}
```

- Now concrete iterators such as a LinkedListIterator or a TreeIterator can implement the Iterator interface and the client doesn't have to change its code despite change in implementation

Solution 4 cont.

```
public interface List{
  public Iterator iterator();
  …
}
public class LinkedList implements List{
  public Iterator iterator(){
    return new LinkedListIterator();
  }
  private class LinkedListIterator implements
  Iterator{
  }
}
```

What is this!
More to come

- Now any number of structures that support List interface can also return a way to iterate through them without clients being able to access internals at all

Design pattern: Iterator

- **Category –** Behavioral design pattern
- **Intent** – Provide a way to access the elements of a collection sequentially
- **Applicability –** Should be used if...
  - ▫ To access contents of a collection without exposing its internal representation
  - ▫ Support multiple traversals of a collection (with Trees preOrder(),postOrder())
  - ▫ Provide uniform interface for traversing different collections (polymorphic iteration)

Iterator participant descriptions

- **Iterator** – defines interface for accessing and traversing the elements
- **ConcreteIterator** – implements iterator interface and keeps track of current position in traversal
- **AbstractCollection** – defines interface for creating a concrete iterator
- **ConcreteCollection** – implements the iterator method and returns an instance of the proper ConcreteIterator

Iterator UML
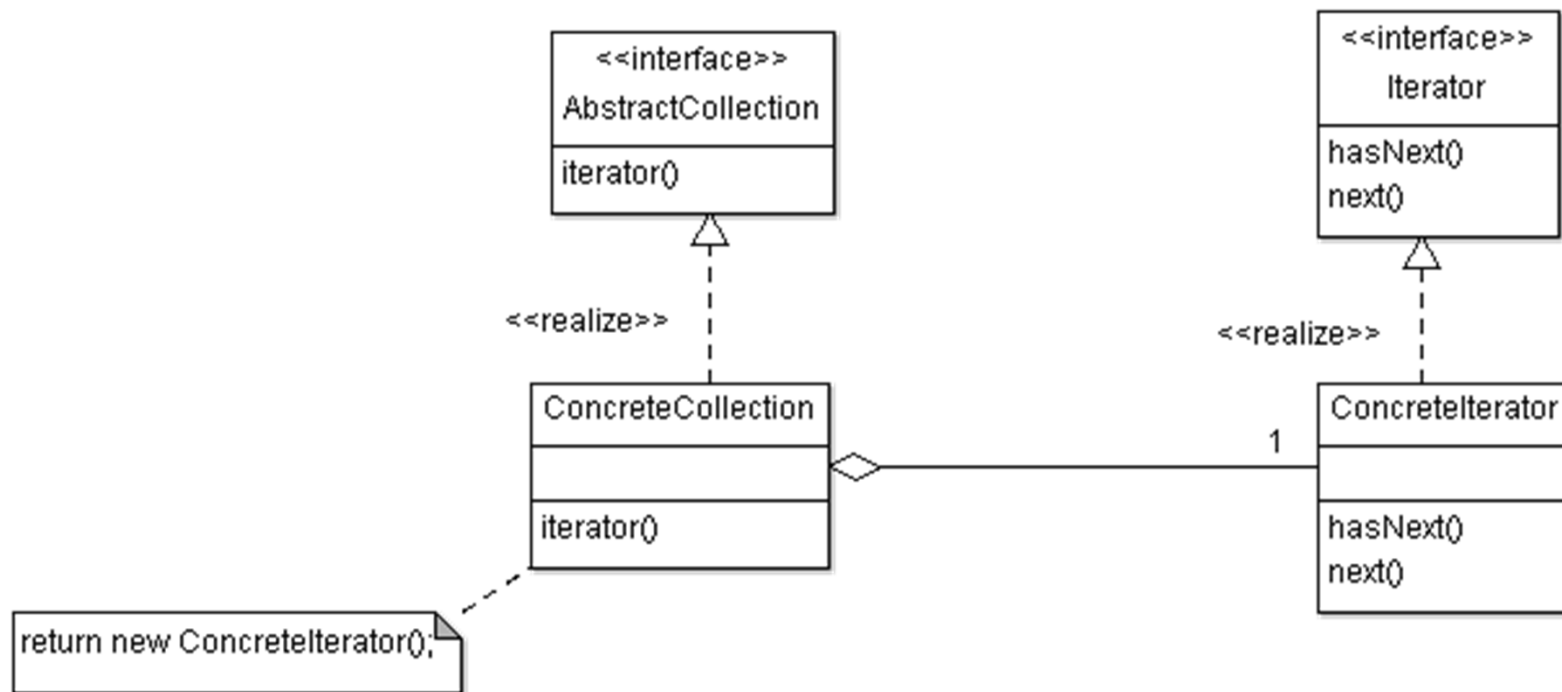
Tree example

- Create UML class diagram extension for an tree data structure support of the iterator pattern.
- Write pseudo code to support iterator pattern for a tree data structure

Group work

- Create UML class diagram extension for an array data structure support of the iterator pattern.
- Write pseudo code to support iterator pattern for an array data structure

Auxiliary classes

- General rule of thumb to be auxiliary class it must only support other classes in the same package

- Two general types of auxiliary classes
  - Classes that support multiple classes
    - Class should be placed into its own file and declared with package visibility
  - Classes that support a single class
    - Inner class

Nested classes

- Java language allows you to declare multiple classes within the same classes

```
public class OuterClass{

    // static nested class

    static class StaticClass{…}

    // Inner class

    class InnerClass{}

}
```

Why nested classes

- Purpose – Like all good OO design, make implementation only visible to those that need it

- It increases encapsulation

- It is a way of logically grouping classes that are only used in one place

- Nested classes can lead to more readable and maintainable code

When to use inner classes

- Used only when a class is used strictly by another class internally

- Allows subclass to access private fields of outer class without exposing them leading to increased encapsulation

- Inner class is declared inside the same class file as the public class

- Visibility of the inner class can be made either private or accessible to subclasses as well

Common auxiliary examples

- Common types of auxiliary classes
  - Iterators
    - Support public interface Iterator but the implementation of the class should not be known by any external class
  - Data structure elements
    - Often for data structures there are components used strictly to maintain the structure that external classes don't need to worry about

Types of classes

- Two kinds of classes
  - Public
    - Reside in their own file with same name as the class
  - *Auxiliary* or *helper* classes
    - Classes used solely for implementing other classes

Static nested classes

- ## It can use all accessibility modifiers

- ## Instances of class can be created independently of parent class

- `OuterClass.StaticNestedClass snc =`

   `new OuterClass.StaticNestedClass();`

- Static attributes can be declared

- Access to OuterClass' static attributes of any visibility

- With reference to OuterClass instance can access attributes of any visibility

Inner classes

- Can use all accessibility modifiers
- Can exist only within the context of an instance of its OuterClass
- `OuterClass.InnerClass innerObject = outerObject.new InnerClass();`
- Only instance attributes can be declared (i.e. no static attributes)
- Can reference any of the parents attributes static or instance regardless of visibility

Which and when

When static nested class

- Class is conceptually tied to Outer class

- Direct access to outer class **instance** unnecessary

- Instance can conceptually exist independently

When inner class

- instances of inner class are somehow conceptually contained by instances of the outer class

- inner class code benefits from direct access to outer class' member

Example static nested class

```
public class LinkedList
  implements List{
    protected Node head, tail;
    static protected class Node{
        Object element;
        Node next, prev;
    }
}
```

Features of design

- Node class is only relevant to LinkedList and enclosed within it

- Nodes exist conceptually outside the context of the LinkedList in terms of references between the Nodes

- Protected class ensures subclasses modify supporting class without violating encapsulation

Example inner class

```
public class LinkedList implements List{
    protected Node head, tail;
    public Iterator iterator(){
        return new LLIterator();
    }
    protected class LLIterator
            implements Iterator{
        Node curNode = head;
        public boolean hasNext(){}
        …
    }
}
```

Features of design

- Iterator tied directly to instance
  - All fields needed can be accessed without exposing fields to outer classes
  - Life of inner instance does not last beyond life of outer class

- Allows implementation of Iterator to be changed without impacting outer classes
- Class can be overridden by subclasses

Nested classes and UML

- 2 takes on whether inner classes should exist in UML

  - Argument for is that it lays out all classes to be developed and dependencies on inner classes from other classes

OuterCla ss	⊗	InnerCla ss
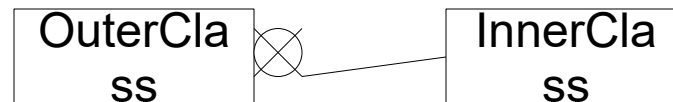
  - Argument against is that it does lay out dependencies by other classes which is against the principle of why it was made an inner class to begin with, instead the relationship should be with the parent class

Group work
Nested classes/inner classes

- Class to support a binary tree

  – Data structure element

  – Get Iterator for in-order traversal

    - Actual code for traversal not needed

- Specifically what type of class for each

- For getIterator what would that method look like, what attributes would be needed on the iterator class

Group work
Nested classes/inner classes

- Writing `Board` object for a game. `Board` made up of many `Tile`s that can be `traverse`d and there is a `State` associated with it that should be initialized at construction of the `Board` class
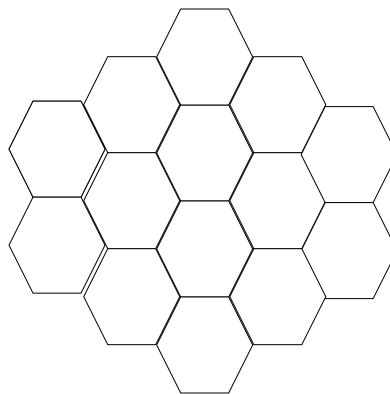
Layout of the board with up to *N* tiles

Collections revisited

- `Collection` framework allows way for groups of elements to be treated uniformly
- Common set of interfaces allows many different concrete data structures to be used interchangeably
  - Collection, Set, SortedSet, List

Iterators of collections

- Regardless of data structure able to traverse all elements in group of elements
- Uniform way through different concrete collections
- Remember from Iterator pattern, allows access to elements while encapsulating implementation

Iterator basics

- All classes in the collections framework support iterator() returning an Iterator
  - hasNext()
  - next()
  - remove()
- Classes that implement the List interface support additional iteration the listIterator() returning a ListIterator

ListIterator

Extends Iterator (hasNext, next, remove)

- hasPrev() – more elements in rev. direction
- nextIndex() – index of element that would be returned by call to next()
- previous() – next element in rev. direction
- previousIndex() – index of element that would be returned by call to previous()
- add(o) – Adds at current position
- set(o) – replaces last element returned

Ordering and sorting

- All classes that depend on an ordering such as SortedSets and SortedMaps depend on there being a defined order to the elements
- 2 ways of defining order in Java

  - the `Comparable` interface to define the *natural order* of elements

  - The `Comparator` interface which can be used to define how a comparison should be done

Natural order

- With Strings/words we consider there to be a natural order anytime you compare two words – alphabetical
- The `Comparable` interface allows you to specify how any two objects should be compared

```
public interface Comparable{

   public int compareTo(Object o);

}
```

compareTo()

- Compares `this` to the passed in object and returns if `this` is "higher/lower" than the passed object
- Result:
  - <0, if `this` is before the passed object
  - 0, if the order is equivalent
  - >0, if the passed object is before `this`

compareTo total order

In order to return predictable results, the compare order must be consistent in both directions

a.compareTo(b)>0 implies b.compareTo(a)<0

a.compareTo(b)<0 implies b.compareTo(a)>0

a.compareTo(b)=0 implies b.compareTo(a)=0

Also:

a.equals(b) is true if and only if a.compareTo(b) is 0

User defined orders

- For many Objects in Java natural order/`Comparable` interface is already defined:
  - Ex. String, Integer, Short

  Others there is none:
  - Ex. List

- `Comparator` interface allows these objects to be compared in a user defined way or objects with a natural order to be compared in a different way

Comparator

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

<0, if o1 precedes o2

0, if the order is equivalent

>0, if o2 precedes o1

Utility methods of `Collections` class

- sort(l) – sorts the list according to natural order
- sort(l, comp) – sorts with passed comparator
- binarySearch(l,k) – binary search assuming list sorted by natural order and returns index or insertion point
- binarySearch(l,k,comp) – same as above assuming sorted by passed comparator
- min(c) – returns minimum of the collection using its natural order
- min(c,comp) – returns minimum of the collection using the passed order
- Reverse(comp) – returns a comparator that reverses the passed comparator