Iterator Pattern

CS342 Fall ‘16

Design problem

Two candy makers use different boxing techniques for their candies. Each have a machine that uses software to scan scan the box, recognize the color and shape of each candy, then map the collection of flavors. The candy makers are now merging, and will be sending out a single box.

Competing Candy Makers

Chocolate is Awesome Candy Makers

- Stores list of candy in a standard array
- Has 15 pieces of candy per candy box

Vanilla is Underrated Candy Makers

- Stores list of candy in an aggregate Box class she created
- Has 10 pieces of candy per candy box

- Both candy makers are dependant on their software implementation to make their candy
- We need to write software that will merge the two processes so that we can print out a list of candy in each box

Design the Candy Software

- Write code to print out the candy type in each box
  - Assume the candy object has a printCandy() method
  - The Chocolate maker just stores candy in a standard array
  - Assume the Vanilla Box class uses a .getSize() method to get the size of the candy box and a getCandy(index) method to retrieve individual candy information

Example Solution

```
def printCandyBoxContents()

        chocolate.each{|candy|
                candy.printCandy()
        }

        for (i in 0..vanilla.getSize()-1)
                vanilla.getCandy(i).printCandy()
        end
end
```

Is that extensible?

- What happens if we merge with additional candy makers?
  - Are we encapsulating what changes?
- Stepping through array and Box class is different. They each have different implementations.
  - We should be writing to an interface, not an implementation

Combine the loops

- How can we combine the two loops into a single loop?
  - What if we let another class iterate through the objects
- We need to add a method to our classes
  - create getIterator() method
- Create a function that takes an iterator to a Candy object, and loops through with the iterator
  - don’t need two loops, can just call the log function on each candy type

New and Improved with Added Iterator

```
def printCandyBoxContents()
        @candy_type.each{ |c|
            self.log(c.getIterator())
        }
end
```

```
def log(candyIterator)
    while candyIterator.hasNext()
        candyIterator.next().printCandy()
    end
end
```

Sequence of sub-objects

- An aggregate object includes arrays, hashes, or any user implemented collection
- Iterators allow the user to sequence through all sub objects with knowing the details of how the aggregate object stores them

What methods are necessary?

- What methods are ‘must-have’ to iterate over an aggregate object?
  - Must Have: constructor, next(), hasNext()
- What methods are ‘nice-to-have’ when iterating over an aggregate object?
  - Nice To Have: current(), start(), end(), startWith()

The Iterator

- The Java and C++ Iterator classes allows Iteration through any collection
  - ```
    ArrayList list = new ArrayList();
    list.add(“blue”);
    list.add(“red”);
    for(Iterator i = list.iterator(); i.hasNext();)
        System.out.println(“item: “ + i.next());
    ```
    ```

“Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.”

External Iterators

- An object that is external to the object being iterated over
- The iterable relationship is composition

```
class ArrayIterator
      def initialize(array)
            @array = array
            @index = 0
      end
      def has_next?
            @index < @array.length
      end
      def item
            @array[@index]
      end
      def next_item
            value = @array[@index]
            @index += 1
            value
      end
end
```

Using the External Iterator

- Requires two objects
  - The Iterator object
  - The Aggregate object
- Can be used for multiple object types
  - Ruby's duck typing allows you to use the external iterator for lots of different kinds of objects with a cast

```
array = ['red', 'green', 'blue']
i = ArrayIterator.new(array)
while i.has_next?
     puts("item: #{i.next_item}")
end

i = ArrayIterator.new('abc')
while i.has_next?
     puts("item:#{i.next_item.chr}")
end
```

Internal Iterators

- What is a potential problem with external iterators?
  - You need a whole extra class, which introduces more complexity
- What if we can build the iterator into the object as a method?
  - Simplifies iteration
  - We could use ruby blocks!

Ruby Block

- Also known as a *closure*
- The block is similar to a lambda, in that it is a stored procedure, except it is only available to the calling, preceding function
- may appear only in the source adjacent to a method call
  - printWord(hello) { |word| puts(word)}

Use Yield for Internal Iterators

- Internal iterators can be created with the yield keyword
- each method of the array class and hash class are not language constructs, but methods written in those classes

Yield Keyword

- Execute a block with the ‘yield’ keyword
  - def printTheWord()

    yield

    end

    printTheWord {puts “The Word”}
- The yield keyword executes the block, then continues running the function

Returning from a Block

- Blocks return from the scope where they are defined
  - ```
    def doWork()
        yield #throws an error
    end

    doWork(){ return 1}
    ```
- If your block is defined in the global scope, and you call return, you get LocalJumpError: unexpected return
    ```

Write the method for the following

Write the class method for the following method call

```
x = [“Hello”, “Goodbye”]
x.each {|x|
   puts(x)
}
```

Pass by Value

- List is still [1,2,3,4] after the each loop
- To change the values in the array use collect!
  - [1,4,9,16]

```
list = [1, 2, 3, 4]
list.each {|x|
    x = x*x    #no change, x is
               #local
}
list.collect! {|x|
    x = x*x    #now x is a
               #reference
}
```

External vs Internal

External Iterator

- Advantages
  - Client drives iteration:
    - only calls next when ready to move onto the next element
  - shared code
- Disadvantages
  - shared code
  - requires separate object

Internal Iterator

- Advantages
  - simplicity
    - no additional objects required
    - no iteration management
      - (calling next)
- Disadvantages
  - iterates through all elements without user control

Enumerable Module

- Add each() to your own aggregate class with the built in enumerable module
  - include Enumerable
- Implement the spaceship operator method override
  - def <=> (other)

    self.value <=> other.value

    end
  - 1 if self>other; 0 if self==other; -1 if self<other

Iterators are not thread safe

- What happens if the array changes during iteration?
- Both Internal and External iterators use an index to keep track
  - indexes will change if an item is deleted or added during iteration
- Make a copy of the array before iterating
  - but this means we cannot modify it
    - This is a design choice based on your collections needs

Classwork: DesignWeaver

Classwork: WakeUp Call