Lessons from the two-three tree Homework

CS 420/520
Andrew P. Black
Goals

- See multiple objects implementing the same interface
- See blocks being used as arguments
  - `replaceMeBy` and `absorb` blocks
  - continuation block as argument to `sort3`
- *Listlessness* as a programming pattern
  - iterators deliver their results one-by-one
  - *Listlessness is Better than Laziness* (Wadler, 1984)
Goals

• See multiple objects implementing the same interface
• See blocks being used as arguments
  • replaceMeBy and absorb blocks
• continuation block as argument to sort3
• Listlessness as a programming pattern
  • iterators deliver their results one-by-one
• Listlessness is Better than Laziness (Wadler, 1984)
• Program to an Interface, not to an Implementation

• The implementation was given; all you had to do was figure out the interface

• Reading tests and documentation to discover the interface

• Resolving ambiguities:
  – writing tests, asking questions
  – spotting bugs or inconsistencies
Using multiple Classes
A student wrote:

I had experience coding a 2-3 tree in CS 163. Back in those days, I struggled for many days to deal with insert and remove. I wrote a 2-page method to add a new node to tree. I used an if-then-else statement to find out if the current node was empty, contained one value, or contained two. And then another nested if inside each branch to see if we needed to add left/middle/right, or go left/middle/right. That was a mess. I could imagine how hard it would be for a person to comprehend the code.

Using OOP to implement it makes life easier. We don’t need to find out which kind of node we are in: we already know. We also already know when we should change to another kind of node, and which it should be. All we need do is implement a specific case in each class, and then let the objects do their jobs.
Using multiple Classes

• A student wrote*:

I had experience coding a 2-3 tree in CS 163. Back in those days, I struggled for many days to deal with insert and remove. I wrote a 2-page method to add a new node to tree. I used an if-then-else statement to find out if the current node was empty, contained one value, or contained two. And then another nested if inside each branch to see if we needed to add left/middle/right, or go left/middle/right. That was a mess. I could imagine how hard it would be for a person to comprehend the code.

Using OOP to implement it makes life easier. We don’t need to find out which kind of node we are in: we already know. We also already know when we should change to another kind of node, and which it should be. All we need do is implement a specific case in each class, and then let the objects do their jobs.

* text corrected for grammar
Using multiple Classes

• A student wrote*:

I had experience coding a 2-3 tree in CS 163. Back in those days, I struggled for many days to deal with insert and remove. I wrote a 2-page method to add a new node to tree. I used an if-then-else statement to find out if the current node was empty, contained one value, or contained two. And then another nested if inside each branch to see if we needed to add left/middle/right, or go left/middle/right. That was a mess. I could imagine how hard it would be for a person to comprehend the code.

Using OOP to implement it makes life easier. We don’t need to find out which kind of node we are in: we already know. We also already know when we should change to another kind of node, and which it should be. All we need do is implement a specific case in each class, and then let the objects do their jobs.

* text corrected for grammar
Using multiple Classes

- A student wrote*:

I had experience coding a 2-3 tree in CS 163. Back in those days, I struggled for many days to deal with insert and remove. I wrote a 2-page method to add a new node to tree. I used an *if-then-else* statement to find out if the current node was empty, contained one value, or contained two. And then another nested *if* inside each branch to see if we needed to add left/middle/right, or go left/middle/right. That was a mess. I could imagine how hard it would be for a person to comprehend the code.

Using OOP to implement it makes life easier. We don’t need to find out which kind of node we are in: we already know. We also already know when we should change to another kind of node, and which it should be. All we need do is implement a specific case in each class, and then let the objects do their jobs.

- Multiple *token* classes in the *glob* homework

- Many different kinds of *component* on a canvas

* text corrected for grammar
Using multiple Classes

• A student wrote*:

I had experience coding a 2-3 tree in CS 163. Back in those days, I struggled for many days to deal with insert and remove. I wrote a 2-page method to add a new node to tree. I used an if-then-else statement to find out if the current node was empty, contained one value, or contained two. And then another nested if inside each branch to see if we needed to add left/middle/right, or go left/middle/right. That was a mess. I could imagine how hard it would be for a person to comprehend the code.

Using OOP to implement it makes life easier. We don’t need to find out which kind of node we are in: we already know. We also already know when we should change to another kind of node, and which it should be. All we need do is implement a specific case in each class, and then let the objects do their jobs.

* text corrected for grammar

- Multiple token classes in the gloo homework
- Many different kinds of component on a canvas
- Many different test cases in a test suite
• When using the state pattern

Use More Objects

• A student writes:

I was pretty happy with my code before I posted a question to the class forum about the behavior of extra symbols inside brackets. At that point I had only one "bracketParseState", rather than the "leftBracketParseState" and "bracketCharsParseState" I ended up with. I chose to raise an error any time one of the other symbols appeared inside brackets, if a left bracket appeared without a right bracket, or if a right bracket appeared before a left bracket.

After reading the discussion on the forum, I switched to the two state implementation, where one is used when a left bracket first appears, and the second one is used to fill the brackets with characters. Now the combinations of symbols described above are all treated as plain characters rather than raising errors. I'm happier with this version of the code.
• Dictionary itself!

• hash-table implementation

• search-tree implementation
Objects have *Two* Interfaces

1. Interface to **use** the object:

   \[
   \textbf{type} \ \text{Dictionary} = \textbf{interface} \ \{
   \text{at}(_) \text{put}(_); \ \text{keys}; \ \text{iterator}; \ \text{do}(_); \ \ldots \}
   \]

2. Interface to **create** the object:

   \[
   \textbf{type} \ \text{DictionaryFactory} = \textbf{interface} \ \{
   \text{dictionary}(_) ; \ \text{dictionary}.\text{withAll}(_) ;
   \text{dictionary} \ \ll ; \ \text{dictionary}.\text{with}(_) ;
   \text{dictionary}.\text{empty} \}
   \]

Assignment wasn’t explicit about this; most students missed its importance.

- To **test** a dictionary, you have to **create** a dictionary
type Collection[T] = type {
    iterator -> Iterator[T]
    // Returns an iterator over my elements. It is an error to modify self while iterating
    // over it. Note: all other methods can be defined using iterator. Iterating over a
    // dictionary yields its values.

    ...
}

    ...
    keys -> Collection[K]  // returns my keys as a lazy sequence in arbitrary order
    values -> Collection[T] // returns my values as a lazy sequence in arbitrary order
    bindings -> Enumerable[ Binding[K, T] ] // returns my bindings as a lazy sequence
}

My tests tell much the same story:

test_small_iterator: <set{3::three, 4::four, 2::two, 1::one, 5::five}>
    should be <set{"five", "three", "two", "one", "four"}>


Simple Methods

• Compare

```python
method ≠(someOtherDictionary) {
    if (self == someOtherDictionary) then {
        return false
    } else {
        return true
    }
}
```

to

```python
method ≠(other) { (self == other).not }
```

• Does `other` have to be a dictionary?
Shop, don’t Build
Shop, don’t Build

- Consider
Shop, don’t Build

• Consider

```ruby
method ++ (t) {
  def newTree = self.copy
  def iter = t.iterator
  var current
  (1 .. iter.zipper.size).do { i →
    current := iter.zipper.at(i)
    (1 .. current.bindingList.size).do { j →
      newTree.at(current.bindingList.at(j).key)
      put (current.bindingList.at(j).value)
    }
  }
  newTree
}
```
Shop, don’t Build

• Consider

```python
method ++ (t) {
    def newTree = self.copy
    def iter = t.iterator
    var current
    (1 .. iter.zipper.size).do { i →
        current := iter.zipper.at(i)
        (1 .. current.bindingList.size).do { j →
            newTree.at(current.bindingList.at(j).key)
                put (current.bindingList.at(j).value)
        }
    }
    newTree
}
```

• Train wreck!

  • This will work *only* when `t` has an *iterator* with a *zipper* method that is itself a collection
Better to reuse the implementation from `collectionsPrelude.dictionary`

```ruby
method ++ (other:Collection[T]) {
    // answers a new dictionary containing all my keys and
    // the keys of other; if other contains one of my keys,
    // other’s value overrides mine

def newDict = self.copy
    other.keysAndValuesDo {k, v ->
        newDict.at(k) put(v)
    }
    return newDict

This works for any other that understands keysAndValuesDo(_)
```

Many of the methods in the dictionary implementation could be factored out into a reusable trait.
Lazy Sequences, aka Streams

- Implementations are available for reuse in `collections.prelude`

```scala
trait iteratorOver[T, R] (sourceIterator: Iterator[T])
  mappedBy (function: Function1[T, R]) => Iterator[R] {
    method asString { "a mapped iterator over {sourceIterator}" }
    method hasNext { sourceIterator.hasNext }
    method next { function.apply(sourceIterator.next) }
  }
```
class lazySequenceOver[T,R] (source: Collection[T])
    mappedBy (function: Function1[T, R]) -> Enumerable[R] {
        use enumerable[T]
        class iterator {
            use iteratorOver[T,R] (source.iterator) mappedBy (function)
        }
        method size { source.size }
        method isEmpty { source.isEmpty }
        method asDebugString { "a lazy sequence mapping over {source}" }
    }
method iteratorOver[T] (sourceIterator: Iterator[T])
  filteredBy(predicate: Predicate1[T]) -> Iterator[T] {
    // returns a trait that supplies the iteration protocol

    var cache
    var cacheLoaded := false
    object {
      method asString { "a filtered iterator over {sourceIterator}" }
      method hasNext {
        // To determine if this iterator has a next element, we have to find
        // an acceptable element; this is then cached, for the use of next
        // If I return true, the cache is loaded.
        if (cacheLoaded) then { return true }
        while { sourceIterator.hasNext } do {
          def outerNext = sourceIterator.next
          def isAcceptable = predicate.apply(outerNext)
          if (isAcceptable) then {
            if (cacheLoaded) then {
              cacheLoaded := true
              cache := outerNext
              return true
            }
          }
        }
        return false
      }
      method next {
        if (hasNext) then {
          cacheLoaded := false
          return cache
        } else {
          IteratorExhausted.raise "no more elements in {self}"
        }
      }
    }
  }
method iteratorOver[T] (sourceIterator: Iterator[T])
    filteredBy(predicate:Predicate1[T]) -> Iterator[T] {
        // returns a trait that supplies the iteration protocol

        var cache
        var cacheLoaded := false
        object {
            method asString { "a filtered iterator over {sourceIterator}" }
            method hasNext {
                // To determine if this iterator has a next element, we have to find
                // an acceptable element; this is then cached, for the use of next
                // If I return true, the cache is loaded.
                if (cacheLoaded) then { return true }
                while { sourceIterator.hasNext } do {
                    def outerNext = sourceIterator.next
                    def isAcceptable = predicate.apply(outerNext)
                    if (isAcceptable) then {
                        cacheLoaded := true
                        cache := outerNext
                        return true
                    }
                }
                return false
            }
            method next {
                if (hasNext) then {
                    cacheLoaded := false
                    return cache
                } else {
                    IteratorExhausted.raise "no more elements in {self}"
                }
            }
        }
    }
method iteratorOver[T] (sourceIterator: Iterator[T])
    filteredBy(predicate:Predicate1[T]) -> Iterator[T] {
        // returns a trait that supplies the iteration protocol
        var cache
        var cacheLoaded := false
        object {
            method asString {
                "a filtered iterator over \{sourceIterator\}" } }
            method hasNext {
                // To determine if this iterator has a next element, we have to find
                // an acceptable element; this is then cached, for the use of next
                // If I return true, the cache is loaded.
                if (cacheLoaded) then { return true }
                while { sourceIterator.hasNext } do {
                    def outerNext = sourceIterator.next
                    def isAcceptable = predicate.apply(outerNext)
                    if (isAcceptable) then {
                        if (cacheLoaded) then {
                            cacheLoaded := true
                            cache := outerNext
                            return true
                        }
                    }
                    return false
                }
            }
        }
        method next {
            if (hasNext) then {
                cacheLoaded := false
                return cache
            } else {
                IteratorExhausted.raise "no more elements in \{self\}"
            }
        }
class lazySequenceOver[T] (source: Collection[T])
    filteredBy(predicate:Predicate1[T]) -> Enumerable[T] {
        use enumerable[T]
        class iterator {
            use iteratorOver[T] (source.iterator) filteredBy (predicate)
        }
        method asDebugString { "a lazy sequence filtering {source}" }
    }
When are you done?

Tests pass

Start
When are you done?

Tests pass

Start

Done
Iterators are tricky to implement

• but handy to use!

• Some languages make it easier, e.g., Python:

```python
def fibonacci(limit):
    # The generator constructs an iterator
    a, b, c = 0, 1, 0
    while c < limit:
        yield a
        # Note: yield, not return
        a, b, c = b, a+b, c+1

it = fibonacci(10)
while True:
    try:
        value = it.__next__()
        # gets the next value; no effect. Also next(it)
    except StopIteration:
        break
    it.__iter__()
    # advances the iterator. Also iter(it)
    print(value)

for v in fibonacci(10):
    # for stmt also uses iterator
    print(v)
```