OO Problem Solving and Testing

A tour of the summer assignment through the eyes of professional programmers

Gregg Berman's crane

In the early 90's I worked with Gregg Berman, a former physicist currently working for the SEC. He wrote their definitive report on the analysis of the flash crash that shook financial markets. We worked for a big hedge fund called Mint Investment Management, which was in Hoboken. Across the street from our offices was a construction lot where, all of one summer, workers were assembling a giant crane. One day, Gregg and I were looking at the giant crane being assembled. There was no visible progress on the building. "You watch," said Gregg. "One Monday, we'll come back and there'll be a whole building there. That's what all of our work is like. We take a long time to build the crane and then, poof, one day there's a building." And he was right.

Steps

- Problem decomposition: break the problem into smaller pieces
- Identify problem specific code and write it along with the appropriate tests
- Find pre-built components in the Java libraries
  - There's no need to test these
- Look for the remainder on the Internet
- Everything we find there must be thoroughly tested

Problem Decomposition

- **Problem specific components**
  - Specific analyses such as computing max price, computing average price, etc.
  - Writing a report to a file
- **Pre-built components**
  - Sorting algorithms in the Collections Framework
  - TreeMap for doing corrections and insertions
- **Reusable components**
  - Managing price records – Will we re-use this format? Let's assume the answer is yes
  - Moving average class
  - Sorting specification classes
  - Convenience methods reading text files
  - Sorting algorithms – We will write our own

PriceRecord class

- We will make a PriceRecord class that understands a String representation of a price record, which is what we have in our text file
- In the constructor, we will do some error checking
- We will also provide a static method for converting a list of String representations of price record into a list of PriceRecord objects
- There are two factory methods for creating comparators – one by date and one by price
  - We will need these for our sort algorithms, or for using the sort algorithms built into the Java collections classes

Why do we need specialized comparators?

- We want our comparators to be non-generic, but we want them to work with any algorithm that compares two price record objects
- As you can see, they both take sort order as a parameter, which allows both ascending and descending sorts
- The adjusted close comparator also takes a tolerance because it will be comparing doubles
- So while, to the outside world, these comparators will look generic, internally, they are doing some things that are related to the specific functionality of the price record class

Dealing with multiple sort criteria

- We have two ways of sorting – BubbleSort and Quicksort
- We have two ways of ordering sorted objects – Ascending and descending
- We have two fields by which to sort – Price or date
- 2 x 2 x 2 = 8
- We could write the sort as 3 nested if statements, with 8 possible resultant calls, but that's a lot of code. It's ugly and hard to test.
- Instead, we're going to make this really elegant...

Polymorphism

- We're going to make sure that all of the sorts we might implement look alike to the outside world
- We're going to do this with the I_SortAlgorithm interface
- Then we're going to make sure that all of our comparators look alike
- In this case all of our comparators will take two price records as arguments to their compare method and will return -1, 0, or 1
- However, what these sorting algorithms and comparators do internally is going to be different depending on the sorting algorithm and comparator we choose
- That allows us to write our code as follows

Static representation of parameters

- What are these specification parameters that we're passing into the comparison factory methods – instances of the SortOrder class?
- We do not want to represent parameters as values when we know they have a limited number of states
- We want to give users of our code the ability to choose those states directly, which is especially useful when you have text completion as we do in the Eclipse IDE
- This greatly reduces the number of mistakes
- See how the constructors are implemented? They're private
- Only the class itself can create objects of this class
- In the case of SortOrder, it creates two objects – one to represent an ascending sort and one to represent a descending sort
- Internally, it can store a multiplier that we then use inside the comparator

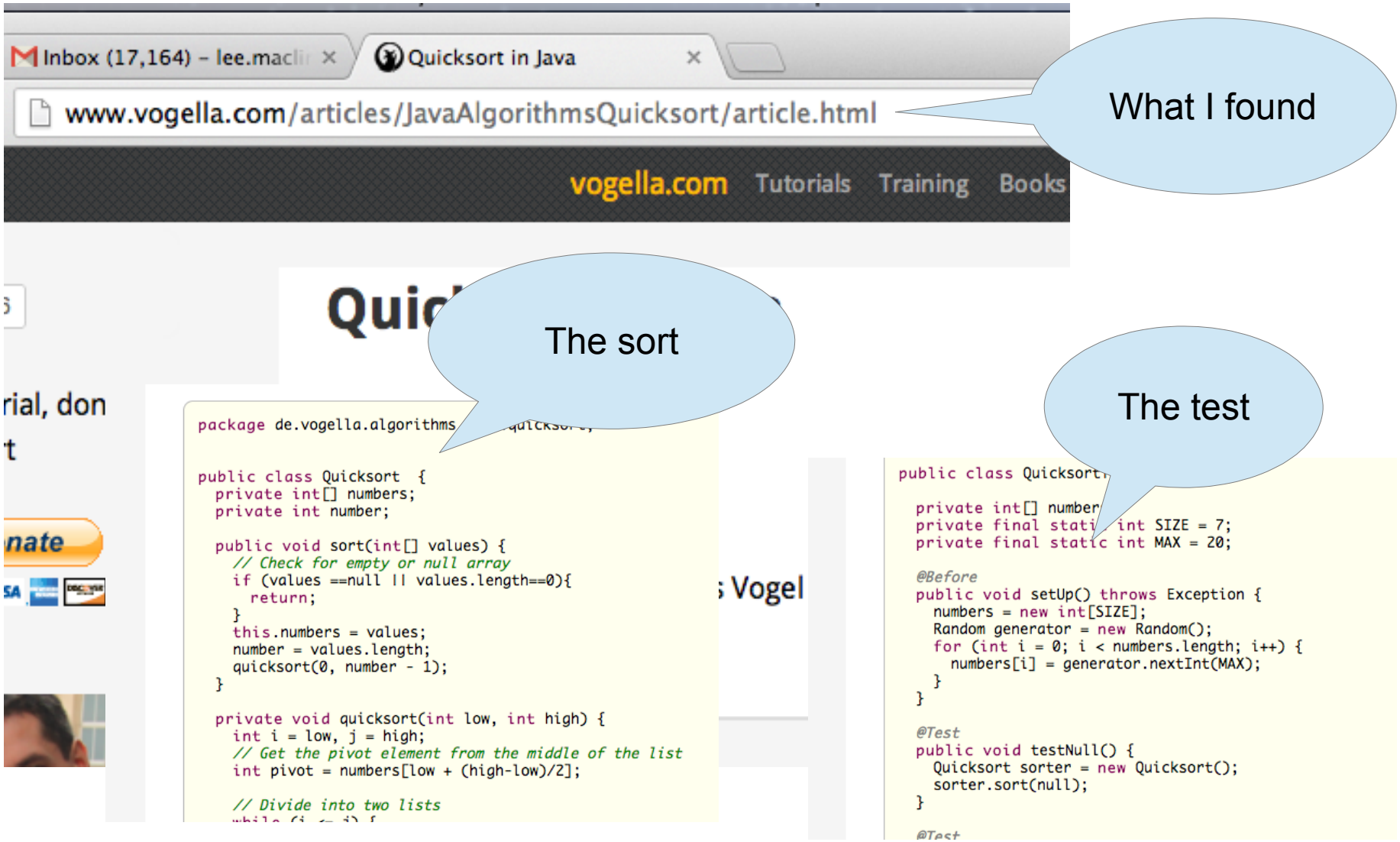
What about more complex sorts?

- What if we wanted to sort descending based on date but – for a given day – ascending based on price?
- Any complex sort can be expressed as a series of less complex sorts
- We already have one reusable class, SortOrder. We will now make another one called CompoundComparator
- It's a comparator that holds a list of other comparators
- For each value, it traverses the list asking each comparator to perform a comparison
- If any comparison is not zero, it exits and returns the result, but if all comparisons return zero, it returns zero
- This allows us to write sorts as follows

Finding a sort algorithm

- There are sorts in the Java Collections Framework
- We will ignore them for the sake of exploring how we'd go about solving a problem for which the code did not conveniently exist – already tested – in our library
- There are a million versions of both the bubble sort and the quick sort on the Internet
- The key is not to wind up using someone's crappy, untested, badly designed code
- First, quicksort...

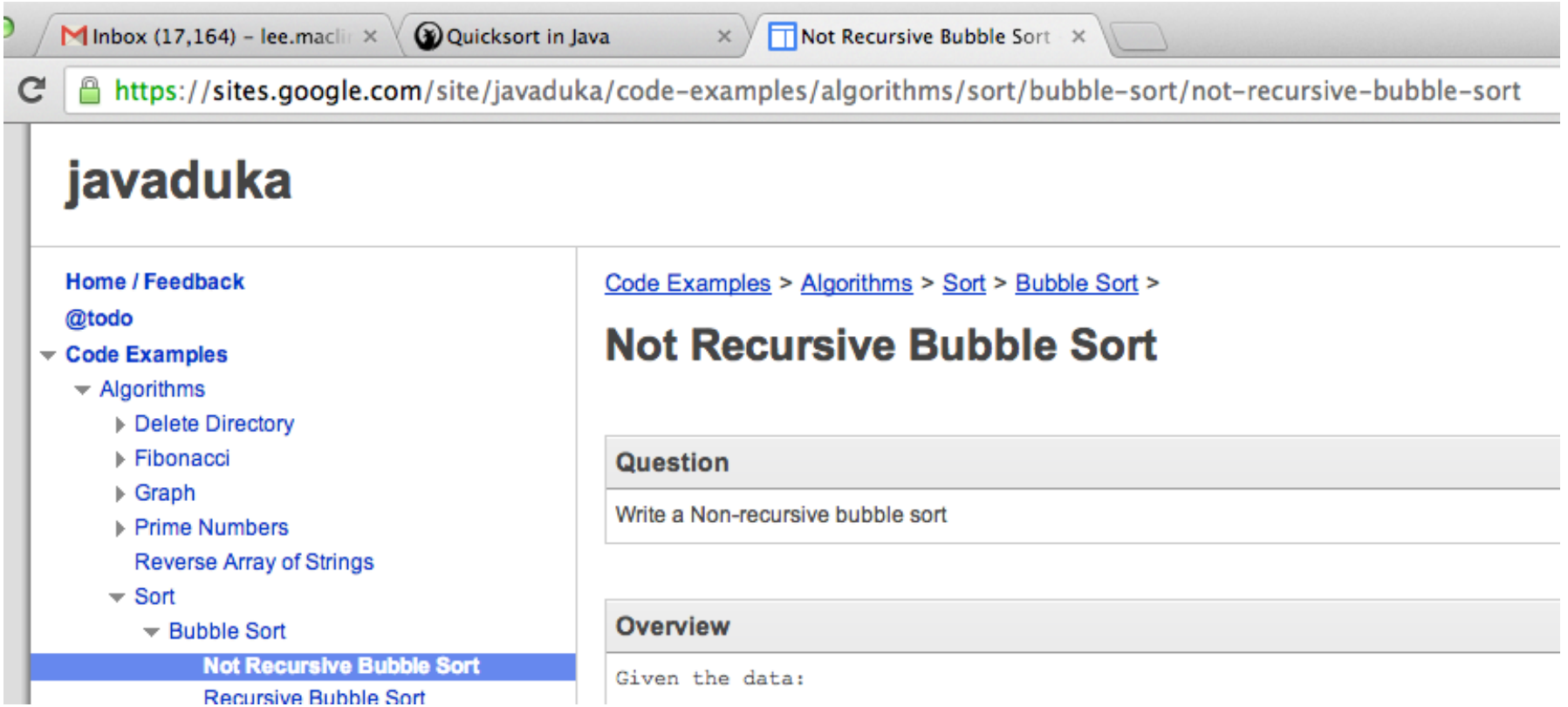
Finding a sort algorithm



It has a couple of problems

- A non-recursive quicksort is a little bit faster
- Some of his tests are stupid
  - Why would you allow a null value to be passed to your sort? We want that to throw an exception
- This thing is not generic – It cannot be used to sort price records, only integers
- I ran the test anyway to make sure it works
- The first thing I then did is rewrote it to be generic: it now takes an array of unknown values and an unknown comparator

I then found a bubble sort



Testing both together

- First, I rewrote the bubble sort to be generic
- I then wrote a test for both the bubble sort and the quicksort as follows
- These are essentially the same tests that I got off the web site but written for the generic versions of the sorts
- Note the testTheTester method... a little precaution
- Also, look at the random tests
  - If you're working in a production environment and you have a simulation designed to test your probabilistic code, leave it running even while the system is in production
  - Often, you will find a crash due to some special condition that you did not anticipate

Making the sorts really generic

- These sorts should work with any array of objects and any comparator
- In fact, why should the object that is doing the sort have to know about the details of the sort
- Instead, let's make both sorts implement the I_SortAlgorithm interface
- We can make the sort a variable that, if it's a bubble sort will act like a bubble sort and, if it's a quicksort will act like a quicksort
- That gives us more room to bring in other sorts later

Let's move on to text file reading

- We wrote our PriceRecord class so that it can be instantiated form a String representation of a price record
- We then gave it a static method that converts a list of String representations of price records to actual PriceRecord objects
- So now we need a class that reads a text file into memory, into a list of String objects
- To our growing collection of reusable classes, we add TextFileReader as follows
- Note that in testing TextFileReading we use temporary files as follows
- Text file reading in DataHandler is now a few lines of code all unit tested as separate functionality

More reuse – MovingAverage

- We now have the following reusable classes and interfaces
  - PriceRecord
  - CompoundComparator
  - SortOrder
  - TextFileReader
  - GenericBubbleSort
  - GenericQuicksort
  - I_SortAlgorithm
- We want to add another – MovingAverage – because computing a moving average is so common

MovingAverage

- What is a moving average calculator?
- It's a queue of some maximum length
- When it's full, we want to calculate a moving average for all the elements in the queue
- But as we add more elements, we want to remove elements from the back of the queue
- Which data structure can do that efficiently? A linked list.
- Clearly, we can derive a moving average from a linked list as follows (see example)
- However, we now have a problem...

Inheritance problems

- By deriving functionality from LinkedList, we've exposed all of its methods to the outside world
- We don't want people to be able to go in and access the methods of LinkedList through our MovingAverage object
- That would mess up our management of the queue that we use to compute the moving average
- How do we limit what methods people can see?
- Instead of a constructor, we can use a static factory method to create an instance of MovingAverage that is returned as an object implementing a simplified interface, I_Moving Average
- We keep the constructor of MovingAverage private to prevent misuse

What's the right data structure for keeping price records in memory?

- We asked you to use an array or list but would TreeMap be more appropriate?
- I used TreeMap to do price corrections – insertions and overwrites, but I immediately converted back to an array
- In my Junit class *Test_DataHandler*, the last method is *testTreeMapVersion,* in which I demonstrate how we could have used a TreeMap instead of an array or list
- As you can see, finding a range of records by date is really easy and happens to be more efficient than a linear search
  - Of course, for records in an array sorted by date, we can always do a binary search
- Insertions into an array are also inefficient, so if we were doing many corrections, a TreeMap may be more efficient
- If we are constantly switching between date sorts and price sorts, we would have to maintain multiple maps – but that wasn't the case

We now have everything we need

- We are ready to look at our DataHandler class
- We will pay particular attention to its testing and we will use a little trick called mocking
- Mock objects are objects of classes derived from the classes we are testing but with special methods that allow us to set up a test state and see whether the code behaves as expected
- For example, in the first unit test of data handler, we instantiate a new object of class DataHandler1, which is derived from DataHandler
- Instead of reading lines from a test file, this object allows us to set the lines that are returned when DataHandler calls getPriceRecords
- By overriding the functionality of the parent class in a temporary derived class – we can set up a very specific state for the object that we want to test and then see how it behaves when we call a method