Chapter 14: Refactoring with Tool Support

by John Brant and Don Roberts
University of Illinois and Urbana-Champaign

In this book, I've concentrated on manual refactoring that anyone with a regular development environment can do. This is not the way that belive refactoring will work in the future. In the future we will have tools that carry out many of the refactorings. As I've said earlier, such a tool already exists in Smalltalk. So for this chapter I thought I'd give you a glimpse of what refactoring is like when you have a good tool. As this tool only exists for Smalltalk, I can only do this with a Smalltalk example. So if you don't understand Smalltalk this chapter may not be as useful for you. But I hope it will give you a flavor of what this kind of tool can do.

The program comes from a thread in the newsgroup comp.lang.smalltalk. An assertion had been made that refactoring is a better way to clarify a complicated program than commenting. A challange was made for a program to act as a demonstration. So someone posted the following program, recoded directly from a published algorithm. One of the properties claimed for this algorithm is that it is much faster than the built in sort capability in Smalltalk.

Once the program was posted John Brant, one of the authors of the Refactoring Browser, posted this response, showing how refactoring could clarify even a messy program such as this. I've decided to leave John's text as it is, as I think he describes things better than I could. His partner on the work for the Smalltalk Browser is Don Roberts, and Don has added comments in the side frames.

- - Martin Fowler

The Starting Program

```
Object subclass: #Sort
  instanceVariableNames:
    'data i j ii ij il iu k l m t tt '
  classVariableNames: ''
  poolDictionaries: ''

Sort class methods

array: anArray
        "Answer a sorted array."
    ^self new sort: anArray size: anArray size.!

condense: anArray size: anInteger
        "Answer sorted array removing nils.
         Parameter size: is without nils."
    | index element array n |
    n := anInteger.
    array := Array new: n.
    index := anArray size.
    [index > 0]
        whileTrue: [
            (element := anArray at: index) == nil
                ifFalse: [array at: n put: element.
                          n := n - 1].
            index := index - 1].
    ^self new sort: array size: anInteger

Help
^'ACM #347 by Singleton, the fastest known sort, was
converted to Smalltalk with minimal changes.

Answers sorted array in ascending order.  Send result array
the message "reversed" to get a descending answer.

The execute: method implements "goto", needed for the
algorithm as published, without filling Smalltalk`s call
stack on large arrays.

"Sort array: anArray" returns a sorted array. Nil values
cannot be compared so they must be removed from arrays
before sorting.  Condense: anArray size: aSize removes nils
then sorts the result..'! !
```

```
Sort methods

execute: aLabel
        "Go to."
    | current |
    current := aLabel.
    [current isNil]
        whileFalse: [current := self perform: current]

label1
        "Perform L1."
    ij := i + j // 2.
    t := data at: ij.
    k := i.
    l := j.
    (data at: i) > t
        ifTrue: [self swap: i].
    (data at: j) < t
        ifTrue: [self swap: j.
                 (data at: i) > t
                    ifTrue: [self swap: i]].
    ^#label2.

label2
        "Perform L2."
    l := l - 1.
    [(data at: l) > t]
        whileTrue: [l := l - 1].
    tt := data at: l.
    ^#label3.

label3
        "Perform L3."
    k := k + 1.
    [(data at: k) < t]
        whileTrue:[k := k + 1].
    k <= l
        ifTrue: [data at: l put: (data at: k).
                 data at: k put: tt.
                 ^#label2].
    (l - i) > (j - k)
        ifTrue: [il at: m put: i.
                 iu at: m put: l.
                 i := k]
        ifFalse:[il at: m put: k.
                 iu at: m put: j.
                 j := l].
    m := m + 1.
    ^#label4
```

```
label4
        "Perform L4."
    j - i > 10
        ifTrue: [^#label1].
    i = ii
        ifTrue: [i < j
                ifTrue: [^#label1]].
    i + 1 to: j do:[:n| i := n.
                        t := data at: i.
                        k := i - 1.
                        (data at: k) > t
                            ifTrue: [self label5]].
    m := m - 1.
    m > 0
        ifTrue: [i := il at: m.
                 j := iu at: m.
                 ^#label4].
    ^nil

label5
        "Perform L5."
    [data at: k + 1 put: (data at: k).
    k := k - 1.
    (data at: k) > t]
        whileTrue: [].
    data at: k + 1 put: t

sort: array size: aSize
        "Answer sorted data in ascending order
        using ACM #347 by Singleton."
    | limit stream |
    data := array.
    i := ii := m := 1.
    (j := aSize) > 131071
        ifTrue: [stream := '' asStream.
                 j printOn: stream.
                 limit := stream contents size * 4]
        ifFalse:[limit := 16].
    il := Array new: limit.  "exact limit := j ln // 2 ln"
    iu := Array new: limit.
    self execute: #label4.
    ^data

swap: element
        "Swap data elements."
    data at: ij put: (data at: element).
    data at: element put: t.
    t := data at: ij.
```

The Refactoring

First, I paste in all the code into VisualWorks, and write a test case:

```
TestCase subclass: #SorterTestCase
instanceVariableNames: "
classVariableNames: "
poolDictionaries: "

TestCase>>sort
| collection newCollection |
collection := Object withAllSubclasses collect: [:each | each name].
newCollection := Sort array: collection.
newCollection inject: "
into: [:sum :each |
self should: [sum <= each].
each]
```

Even given sophisticated refactoring tools that can ensure behavior-preservation, there are always transformations that are not behavior-preserving in general, but are behavior-preserving for this particular application.

Therefore, you should always have a set of tests in place before refactoring.

I run the test case "(TestCase selector: #sort) run" and it passes so I go onto refactoring the class.

After browsing the code, I outline some goals:

1. I'd like to remove the execute: method

2. Rename the badly named methods / variables.

Also, I'd like to record what could be done automatically using the Refactoring Browser, and what had to be done manually (on the steps below, I've marked the Refactoring Browser steps with "*[Refactoring Browser]*" and the manual steps with "*[manual]*).

Anyway, on to the refactoring.

I like to start with the class methods to get an idea of how to create the class, so I look at the condense:size: method. I don't see any senders of this method and I don't know how to use it, so I remove it *[Refactoring Browser]*.

Next, I look at the `array:` method. It just forwards itself to the `sort:size:` method. However, the second argument is just dependent on the first

so I'd like to get rid of it. So I go to the sort:size: method and look at all senders, and there is only one. Therefore, I change the sort:size: method into a sort: method (removing the old sort:size: method) *[manual]*:

You do not have to completely understand code to refactor it. In fact, often the act of refactoring will help you understand it. Tests give you the freedom to explore because they will tell you if something breaks.

Code that is hard to understand is often in a poorly factored state. Arranging the code so it is comprehensible is almost always a good thing.

```
sort: array
  "Answer sorted data in ascending order
  using ACM #347 by Singleton."

  | limit stream aSize |
  aSize := array size.
  data := array.
  i := ii := m := 1.
  (j := aSize) > 131071
      ifTrue:
          [stream := WriteStream on: ''.
          j printOn: stream.
          limit := stream contents size * 4]
      ifFalse: [limit := 16].
  il := Array new: limit."exact limit := j ln // 2 ln"
  iu := Array new: limit.
  self execute: #label4.
  ^data
```

and change the sender:

```
array: anArray
  "Answer a sorted array."

  ^self new sort: anArray
```

I then run my test case, and everything still works.

I move on to the sort: method, and start looking at the variables it uses. I see variables i, ii, m, j, il, and iu. Looking at the references to each variable, I see that ii is only assigned to 1 so I change its use to 1 and remove the variable *[manual]*. My tests still work.

Looking at the i & j variables, it appears that they are indices into the collection and i is always less than j. Also, i starts out at 1 and j at the array size. So I guess that i is the startIndex and j is the stopIndex, and

rename the variables. I may be wrong, but I'll rename them later if I am -- right now I just need to understand the algorithm *[Refactoring Browser]*.

I look at the `m`, `il`, and `iu` variables, and couldn't understand them, so I leave them for now.

Once again I look at the `sort:` method. I see that the `aSize` variable really isn't doing much. It is only used once, so we might as well inline it into its reference *[Refactoring Browser]*:

```
sort: array
  "Answer sorted data in ascending order
  using ACM #347 by Singleton."

  | limit stream |
  data := array.
  startIndex := m := 1.
  (stopIndex := array size) > 131071
      ifTrue:
          [stream := WriteStream on: ''.
          stopIndex printOn: stream.
          limit := stream contents size * 4]
      ifFalse: [limit := 16].
  il := Array new: limit."exact limit := j ln // 2 ln"
  iu := Array new: limit.
  self execute: #label4.
  ^data
```

I really don't have any ideas on what the rest of this code is doing, so I move onto another method (`label1`). In this method I see `ij := (startIndex + stopIndex) // 2` which is really the middle index, so I rename `ij` to `middleIndex` *[Refactoring Browser]*.

Next I see that `t` is the value of data at the `middleIndex`, so I'll guess that it should be named `middleElement`, and rename it *[Refactoring Browser]*.

I can't determine what `k` and `l` are so I leave them. Now looking at the rest of the `label1` method, it appears that we are putting the median value of the first, middle, and last elements into the middle index. This is one of the steps in the median of three quick sort algorithm, so I rename label1 to `medianOfThreeQuickSort` *[Refactoring Browser]*. I don't know that it is the median of three quick sort algorithm, but at least we have a better name than `label1`.

I don't see much more that I can do with the medianOfThreeQuickSort method for now, so I move onto the label2 method. The method looks like we are searching backwards until we find an element that is less than the middle element, so I rename it to be findLastElementLessThanMiddleElement, and move onto the label3 method *[Refactoring Browser]*.

Automated low-level refactoring makes these types of manipulations very inexpensive an allows speculation like this.

In Smalltalk, big method names are encouraged and reduce the need for comments

The label3 method is a long method and I'd like to break it up. Looking at the first two lines, it appears that we are searching for some value greater than the middle element. So I extract it to be *[Refactoring Browser]*:

Long methods are a good indicator of places that should be refactored. The rule of thumb is that every method should do 1 thing, and the steps within a method should all be at the same level of abstraction.

```
findFirstElementGreaterThanMiddleElement
  k := k + 1.
  [(data at: k) < middleElement] whileTrue: [k := k + 1]
```

I don't really understand the rest of label3 so I'll leave it for now.

Now, I'd like to remove some of the symbols being returned. Looking at the execute: method it appears that where ever a symbol is returned, it will just perform that symbol, so they can be replaced with self sends. I try to replace all returns with self sends, but when I run my test, I break it after a couple seconds (either I made a mistake or the direct sends are too slow). So I back out of the replacement which is easy since I hadn't made too many changes. However, I'd still like to remove the sends, so I opt to just replace the ones in findLastElementLessThanMiddleElement and medianOfThreeQuickSort. I run my tests, and it still works and isn't noticeably slower *[manual]*.

When changing the findLastElementLessThanMiddleElement, I notice that it is doing more than just finding the last element. I'd like to move this extra behavior into its own methods. First, I look at the tt assignment statement. This variable doesn't appear to be used until we get into the label3 method, so I move it there and run the tests *[manual]*:

```
label3
  "Perform L3."
```

```
self findFirstElementGreaterThanMiddleElement.
k <= l
    ifTrue:
        [tt := data at: l.
        data at: l put: (data at: k).
        data at: k put: tt.
        ^#findLastElementLessThanMiddleElement].
l - startIndex > (stopIndex - k)
    ifTrue:
        [il at: m put: startIndex.
        iu at: m put: l.
        startIndex := k]
    ifFalse:
        [il at: m put: k.
        iu at: m put: stopIndex.
        stopIndex := l].
m := m + 1.
^#label4
```

Now it appears that label3 is just swapping the elements in l & k when k <= l, so I change label3 to use VW's swap:with: method *[manual]*:

```
label3
  "Perform L3."

  self findFirstElementGreaterThanMiddleElement.
  k <= l
      ifTrue:
          [data swap: k with: l.
          ^#findLastElementLessThanMiddleElement].
  l - startIndex > (stopIndex - k)
      ifTrue:
          [il at: m put: startIndex.
          iu at: m put: l.
          startIndex := k]
      ifFalse:
          [il at: m put: k.
          iu at: m put: stopIndex.
          stopIndex := l].
  m := m + 1.
  ^#label4
```

This gets rid of all references to tt, so I remove the instance variable *[Refactoring Browser]*.

Once again, I move back to findLastElementLessThanMiddleElement and see that it is still doing more than finding the last element. I'd like to

get rid of this, so the plan is to inline the label3 method, rename the method, and extract the true findLastElementLessThanMiddleElement method.

After inlining the label3 send and removing label3 method I get *[Refactoring Browser]*:

```
findLastElementLessThanMiddleElement
  "Perform L2."

  l := l - 1.
  [(data at: l) > middleElement] whileTrue: [l := l - 1].
  self findFirstElementGreaterThanMiddleElement.
  k <= l
      ifTrue:
          [data swap: k with: l.
          ^#findLastElementLessThanMiddleElement].
  l - startIndex > (stopIndex - k)
      ifTrue:
          [il at: m put: startIndex.
          iu at: m put: l.
          startIndex := k]
      ifFalse:
          [il at: m put: k.
          iu at: m put: stopIndex.
          stopIndex := l].
  m := m + 1.
  ^#label4
```

Since I don't know what its purpose is yet, I rename it to be unknown *[Refactoring Browser]*. Now, I can extract the first two lines as findLastElementLessThanMiddleElement *[Refactoring Browser]*:

```
findLastElementLessThanMiddleElement
  l := l - 1.
  [(data at: l) > middleElement] whileTrue: [l := l - 1]

and:

unknown
  "Perform L2."

  self findLastElementLessThanMiddleElement.
  self findFirstElementGreaterThanMiddleElement.
  k <= l
```

```
        ifTrue:
            [data swap: k with: l.
            ^#unknown].
    l - startIndex > (stopIndex - k)
        ifTrue:
            [il at: m put: startIndex.
            iu at: m put: l.
            startIndex := k]
        ifFalse:
            [il at: m put: k.
            iu at: m put: stopIndex.
            stopIndex := l].
    m := m + 1.
    ^#label4
```

Now, I'd like to understand more about the unknown method, so first I inline the whileTrue: dispatching loop from execute: into the unknown method *[manual]*:

```
unknown
    "Perform L2."


    [self findLastElementLessThanMiddleElement.
    self findFirstElementGreaterThanMiddleElement.
    k <= l]
            whileTrue: [data swap: k with: l].
    l - startIndex > (stopIndex - k)
        ifTrue:
            [il at: m put: startIndex.
            iu at: m put: l.
            startIndex := k]
        ifFalse:
            [il at: m put: k.
            iu at: m put: stopIndex.
            stopIndex := l].
    m := m + 1.
    ^#label4
```

Since this was a manual step, I run my tests and they worked.

Now remembering some about the quicksort algorithm, I remember that there was a partitioning step, then we recursively sorted the partitions (of course this assumes that my guess of this being a quicksort is correct -- but so far it still matches). Looking at the unknown method,

it appears that the whileTrue: look is doing the partitioning so I extract it to a partition method *[Refactoring Browser]*:

```
partition

  [self findLastElementLessThanMiddleElement.
  self findFirstElementGreaterThanMiddleElement.
  k <= l]
          whileTrue: [data swap: k with: l]

and:

unknown
  "Perform L2."

  self partition.
  l - startIndex > (stopIndex - k)
      ifTrue:
          [il at: m put: startIndex.
          iu at: m put: l.
          startIndex := k]
      ifFalse:
          [il at: m put: k.
          iu at: m put: stopIndex.
          stopIndex := l].
  m := m + 1.
  ^#label4
```

Having had as much fun as I can stand with the methods, I move onto trying to eliminate / rename some of the instance variables. I look at k & l in the medianOfThreeQuickSort method. These variables aren't used by this method, and I'd like to move them where they're used. After further investigation, it appears that I can move them to the unknown method *[manual]*:

```
unknown
  "Perform L2."

  k := startIndex.
  l := stopIndex.
  self partition.
  l - startIndex > (stopIndex - k)
      ifTrue:
          [il at: m put: startIndex.
          iu at: m put: l.
          startIndex := k]
      ifFalse:
```

```
            [il at: m put: k.
            iu at: m put: stopIndex.
            stopIndex := l].
    m := m + 1.
    ^#label4

and:

medianOfThreeQuickSort
    "Perform L1."

    middleIndex := (startIndex + stopIndex) // 2.
    middleElement := data at: middleIndex.
    (data at: startIndex) > middleElement ifTrue: [self swap: startIndex].
    (data at: stopIndex) < middleElement
        ifTrue:
            [self swap: stopIndex.
            (data at: startIndex) > middleElement ifTrue: [self swap: startIndex]].
    ^self unknown
```

I re-run the tests and everything still works, so I'm happy. While I wasn't able to remove them, at least they are closer to where they're being used.

I now move onto the `label4` method. I'd like to remove the indirect message send of `#label4` at the end of the method. I use the same inlining `whileTrue:` from the execute: as I did before *[manual]*:

```
label4
    "Perform L4."


    [stopIndex - startIndex > 10 ifTrue: [^#medianOfThreeQuickSort].
    startIndex = 1
        ifTrue: [startIndex < stopIndex ifTrue: [^#medianOfThreeQuickSort]].
    startIndex + 1 to: stopIndex
        do:
            [:n |
            startIndex := n.
            middleElement := data at: startIndex.
            k := startIndex - 1.
            (data at: k) > middleElement ifTrue: [self label5]].
    m := m - 1.
    m > 0]
            whileTrue:
               [startIndex := il at: m.
               stopIndex := iu at: m].
    ^nil
```

Now looking at m, il, and iu, I notice that they are always used together. m is the index into the array and il is used to assign the startIndex and iu is used for the stopIndex. Remembering back in my algorithms book, I remember a variant of the quicksort that wasn't directly recursive and used a stack of start and stop indices, so I assume that these variables are for that. I rename il to startIndexStack and iu to stopIndexStack. Now since these represent stacks, I'd prefer to use an OrderedCollection to represent them instead of arrays that we must precompute the size of. I then change the sort: and label4 methods *[manual]*:

```
sort: array
  "Answer sorted data in ascending order
  using ACM #347 by Singleton."

  | limit stream |
  data := array.
  startIndex := 1.
  (stopIndex := array size) > 131071
      ifTrue:
          [stream := WriteStream on: ''.
          stopIndex printOn: stream.
          limit := stream contents size * 4]
      ifFalse: [limit := 16].
  startIndexStack := OrderedCollection new.
  stopIndexStack := OrderedCollection new.
  self execute: #label4.
  ^data

and:

label4
  "Perform L4."


  [stopIndex - startIndex > 10 ifTrue: [^#medianOfThreeQuickSort].
  startIndex = 1
      ifTrue: [startIndex < stopIndex ifTrue: [^#medianOfThreeQuickSort]].
  startIndex + 1 to: stopIndex
      do:
          [:n |
          startIndex := n.
          middleElement := data at: startIndex.
          k := startIndex - 1.
          (data at: k) > middleElement ifTrue: [self label5]].
  startIndexStack isEmpty not]
```

```
        whileTrue:
           [startIndex := startIndexStack removeLast.
           stopIndex := stopIndexStack removeLast].
  ^nil
```

I then run the tests, and they fail. Looking at the failure, I notice that I forgot to change unknown method for our new representation.

Changing unknown and rerunning the tests work:

```
unknown
  "Perform L2."

  k := startIndex.
  l := stopIndex.
  self partition.
  l - startIndex > (stopIndex - k)
      ifTrue:
          [startIndexStack add: startIndex.
          stopIndexStack add: l.
          startIndex := k]
      ifFalse:
          [startIndexStack add: k.
          stopIndexStack add: stopIndex.
          stopIndex := l].
  ^#label4
```

Now I remove m since it isn't used anymore *[Refactoring Browser]*.

I also notice that the special case code for size > 131071 isn't really necessary anymore since we have changed to OrderedCollections, so I remove the check *[manual]*:

```
sort: array
  "Answer sorted data in ascending order
  using ACM #347 by Singleton."

  data := array.
  startIndex := 1.
  stopIndex := array size.
  startIndexStack := OrderedCollection new.
  stopIndexStack := OrderedCollection new.
  self execute: #label4.
  ^data
```

Once again I start looking at the variables. Every variable except k & l has a decent name. I'd like to give them good names also. l appears to be associated with findLastElementLessThanMiddleElement and k with findFirstElementGreaterThanMiddleElement. I rename l to be lastIndexLessThanMiddleElement and k to firstIndexGreaterThanMiddleElement *[Refactoring Browser]*.

Growing tired of that work, I move to something new, the swap: method. There's not much there, but I decide to use VW's swap:with: method instead *[manual]*:

```
swap: element
  "Swap data elements."

  data swap: middleIndex with: element.
  middleElement := data at: middleIndex
```

Since there isn't a whole lot more I can do here, I move back to the label4 method. For many sorts, it appears that it does a median of three quicksort, but there is a special case where it does looping. Looking more closely at the loop, it appears that it compares the current element with the previous element, and whenever it finds one that is less than the previous then it calls label5 method. This appears to be an insertion sort, so I extract the code into an insertionSort method *[Refactoring Browser]*:

```
insertionSort
  startIndex + 1 to: stopIndex
      do:
          [:n |
          startIndex := n.
          middleElement := data at: startIndex.
          firstIndexGreaterThanMiddleElement := startIndex - 1.
          (data at: firstIndexGreaterThanMiddleElement) > middleElement
             ifTrue: [self label5]]

and:

label4
  "Perform L4."
```

```
[stopIndex - startIndex > 10 ifTrue: [^#medianOfThreeQuickSort].
startIndex = 1
    ifTrue: [startIndex < stopIndex ifTrue: [^#medianOfThreeQuickSort]].
self insertionSort.
startIndexStack isEmpty not]
        whileTrue:
           [startIndex := startIndexStack removeLast.
           stopIndex := stopIndexStack removeLast].
^nil
```

Looking at label4, there appears to be a special case when startIndex = 1. I'd prefer to get rid of this special case and just use the insertion sort. I comment out the code, and run the tests, but they fail in label5. After closer inspection of the failure, it appears that the insertion sort routine will walk off the beginning of the array. I change the insertion sort to check for this condition and rerun the tests (they worked) *[manual]*:

```
label5
  "Perform L5."


  [data at: firstIndexGreaterThanMiddleElement + 1
      put: (data at: firstIndexGreaterThanMiddleElement).
  firstIndexGreaterThanMiddleElement := firstIndexGreaterThanMiddleElement
          - 1.
  firstIndexGreaterThanMiddleElement > 0
      and: [(data at: firstIndexGreaterThanMiddleElement) > middleElement]]
          whileTrue: [].
  data at: firstIndexGreaterThanMiddleElement + 1 put: middleElement

and:

label4
  "Perform L4."


  [stopIndex - startIndex > 10 ifTrue: [^#medianOfThreeQuickSort].
  self insertionSort.
  startIndexStack isEmpty not]
          whileTrue:
             [startIndex := startIndexStack removeLast.
             stopIndex := stopIndexStack removeLast].
  ^nil
```

I change label4 to send the medianOfThreeQuickSort message directly and change the test to be ifTrue:ifFalse: *[manual]*:

```
label4
  "Perform L4."


  [stopIndex - startIndex > 10
      ifTrue: [^self medianOfThreeQuickSort]
      ifFalse: [self insertionSort].
  startIndexStack isEmpty not]
          whileTrue:
             [startIndex := startIndexStack removeLast.
             stopIndex := stopIndexStack removeLast].
  ^nil
```

I decide that I've had enough of the bad label4 name, and rename it to be sort since it is the top level method that does the sorting *[Refactoring Browser]*.

Now I'd like to get rid of the return of "self medianOfThreeQuickSort" in the sort method and just let the processing fall on through. Since medianOfThreeQuickSort always returns #sort, we could just let sort handling the processing. My first attempt just removed the return, however, this caused my test cases to fail. After a couple minutes thinking about the problem, I remembered that we need to sort both halves of the partition for quick sort, and the stack was only holding one half. I changed the unknown method to add both partitions to the stacks *[manual]*:

```
unknown
  "Perform L2."

  firstIndexGreaterThanMiddleElement := startIndex.
  lastIndexLessThanMiddleElement := stopIndex.
  self partition.
  lastIndexLessThanMiddleElement - startIndex
      > (stopIndex - firstIndexGreaterThanMiddleElement)
          ifTrue:
             [startIndexStack
                add: startIndex;
                add: firstIndexGreaterThanMiddleElement.
             stopIndexStack
                add: lastIndexLessThanMiddleElement;
                add: stopIndex]
          ifFalse:
             [startIndexStack
                add: firstIndexGreaterThanMiddleElement;
```

```
              add: startIndex.
            stopIndexStack
              add: stopIndex;
              add: lastIndexLessThanMiddleElement].
  ^#sort
```

I rerun the test cases, and everything works so I'm happy...

Now we're ready to remove the execute: method that has been causing us so much headache. I change the sort: method to send the message directly and test *[manual]*:

```
sort: array
  "Answer sorted data in ascending order
  using ACM #347 by Singleton."

  data := array.
  startIndex := 1.
  stopIndex := array size.
  startIndexStack := OrderedCollection new.
  stopIndexStack := OrderedCollection new.
  self sort.
  ^data
```

Since the execute: method is no longer used, I remove it *[Refactoring Browser]*. Also, since the return values of sort and unknown aren't used I remove them also *[manual]*. My tests still run...

I still need to name unknown and label5 methods, so I look at the unknown method. First, I notice that the first two assignment statements can be moved into the partition method, so I do that *[manual]*:

```
partition
  firstIndexGreaterThanMiddleElement := startIndex.
  lastIndexLessThanMiddleElement := stopIndex.

  [self findLastElementLessThanMiddleElement.
  self findFirstElementGreaterThanMiddleElement.
  firstIndexGreaterThanMiddleElement <= lastIndexLessThanMiddleElement]
          whileTrue:
            [data swap: firstIndexGreaterThanMiddleElement
              with: lastIndexLessThanMiddleElement]
```

```
and:

unknown
  "Perform L2."

  self partition.
  lastIndexLessThanMiddleElement - startIndex
      > (stopIndex - firstIndexGreaterThanMiddleElement)
          ifTrue:
             [startIndexStack
               add: startIndex;
               add: firstIndexGreaterThanMiddleElement.
             stopIndexStack
               add: lastIndexLessThanMiddleElement;
               add: stopIndex]
          ifFalse:
             [startIndexStack
               add: firstIndexGreaterThanMiddleElement;
               add: startIndex.
             stopIndexStack
               add: stopIndex;
               add: lastIndexLessThanMiddleElement]
```

It looks like the last `ifTrue:ifFalse:` statement is setting up the recursion, so I extract it into a `recursivelySortPartitions` method *[Refactoring Browser]*.

```
recursivelySortPartitions
  lastIndexLessThanMiddleElement - startIndex
      > (stopIndex - firstIndexGreaterThanMiddleElement)
          ifTrue:
             [startIndexStack
               add: startIndex;
               add: firstIndexGreaterThanMiddleElement.
             stopIndexStack
               add: lastIndexLessThanMiddleElement;
               add: stopIndex]
          ifFalse:
             [startIndexStack
               add: firstIndexGreaterThanMiddleElement;
               add: startIndex.
             stopIndexStack
               add: stopIndex;
               add: lastIndexLessThanMiddleElement]

and:

unknown
```

```
  "Perform L2."

  self partition.
  self recursivelySortPartitions
```

The unknown method is rather short, so I'll inline it into its sender and remove it *[Refactoring Browser]*:

```
medianOfThreeQuickSort
  "Perform L1."

  middleIndex := (startIndex + stopIndex) // 2.
  middleElement := data at: middleIndex.
  (data at: startIndex) > middleElement ifTrue: [self swap: startIndex].
  (data at: stopIndex) < middleElement
      ifTrue:
          [self swap: stopIndex.
          (data at: startIndex) > middleElement ifTrue: [self swap: startIndex]].
  self partition.
  self recursivelySortPartitions
```

However, the first part of the method isn't at the same abstraction level as the last two statements, so I'll extract the first part into its own method *[Refactoring Browser]*:

```
pickMedianElement
  middleIndex := (startIndex + stopIndex) // 2.
  middleElement := data at: middleIndex.
  (data at: startIndex) > middleElement ifTrue: [self swap: startIndex].
  (data at: stopIndex) < middleElement
      ifTrue:
          [self swap: stopIndex.
          (data at: startIndex) > middleElement ifTrue: [self swap: startIndex]]

and:

medianOfThreeQuickSort
  "Perform L1."

  self pickMedianElement.
  self partition.
  self recursivelySortPartitions
```

Going back to the recursivelySortPartitions method we can see that one case sorts the first partition then the last and the other case sorts

the last then the first. I extract these into their own methods *[Refactoring Browser]*:

```
recursivelySortLastThenFirstPartition
  startIndexStack
      add: startIndex;
      add: firstIndexGreaterThanMiddleElement.
  stopIndexStack
      add: lastIndexLessThanMiddleElement;
      add: stopIndex

recursivelySortFirstThenLastPartition
  startIndexStack
      add: firstIndexGreaterThanMiddleElement;
      add: startIndex.
  stopIndexStack
      add: stopIndex;
      add: lastIndexLessThanMiddleElement

recursivelySortPartitions
  lastIndexLessThanMiddleElement - startIndex
      > (stopIndex - firstIndexGreaterThanMiddleElement)
          ifTrue: [self recursivelySortLastThenFirstPartition]
          ifFalse: [self recursivelySorFirstThenLastPartition]
```

Now we only have the `label5` method left. It is used by the insertion sort, and I rename it to be `insertSortedElement` *[Refactoring Browser]*.

Looking more at the insertion sort, I'd like to split the variables so that we don't use `middleElement` (which doesn't make sense for an insertion sort). First I try:

```
insertionSort
  startIndex to: stopIndex - 1
      do: [:n | (data at: n - 1) > (data at: n) ifTrue: [self insertSortedElementAt: n]]

and

insertSortedElementAt: anIndex
  | index element |
  element := data at: anIndex + 1.
  index := anIndex.

  [data at: index + 1 put: (data at: index).
  index := index - 1.
  index > 0 and: [(data at: index) > element]]
          whileTrue: [].
```

```
  data at: index + 1 put: element
```

but this failed. I had changed the start and stop, but forgot to change the comparison indices *[manual]*:

```
insertionSort
  startIndex to: stopIndex - 1
      do: [:n | (data at: n) > (data at: n + 1) ifTrue: [self insertSortedElementAt: n]]
```

This works, however I'd still like to get rid of the condition in the `insertionSort` method. The conditional is also present in the `insertSortedElementAt:` method. I reorder the `whileTrue:` stuff to get rid of the condition in the `insertionSort` method *[manual]*:

```
insertionSort
  startIndex to: stopIndex - 1 do: [:n | self insertSortedElementAt: n]

insertSortedElementAt: anIndex
  | index element |
  element := data at: anIndex + 1.
  index := anIndex.
  [index > 0 and: [(data at: index) > element]] whileTrue:
          [data at: index + 1 put: (data at: index).
          index := index - 1].
  data at: index + 1 put: element
```

Finally, I notice that the 0 could be changed to be the start index of the sort in the `insertSortedElementAt:` method. I make the change and retest *[manual]*

```
insertSortedElementAt: anIndex
  | index element |
  element := data at: anIndex + 1.
  index := anIndex.
  [index >= startIndex and: [(data at: index) > element]] whileTrue:
          [data at: index + 1 put: (data at: index).
          index := index - 1].
  data at: index + 1 put: element
```

After all these changes were made, I started to think about improving it further. I was thinking of using object recursion with a quick sort

and insertion sort objects. However, I thought I'd see how my changes had affected performance. I ran my test and it took ~560 ms, and then I tested the standard SortedCollection from VisualWorks, and it only took ~525 ms. Since I had heard about how much faster this version was than VisualWorks' SortedCollection, I decided to load the original version and see how it compared. The original version wasn't any faster, and in fact it was even somewhat slower than my refactored version! Its times were around 570 ms. I then realized that the real refactoring for this problem is to replace all code such as `Sort array: aCollection` with `aCollection asSortedCollection` and remove the Sort class. After this refactoring every line of code left doesn't need commenting -- there are no lines left...

Anyway, if you'd like to see my refactored version of the Sort class before the final remove refactoring, I've attached it below. A little over half of the refactoring steps performed were done by the Refactoring Browser.

```
Object subclass: #Sort
  instanceVariableNames: 'data startIndex stopIndex middleIndex middleElement startIndexStack
stopIndexStack lastIndexLessThanMiddleElement firstIndexGreaterThanMiddleElement '
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Junk'!


!Sort methodsFor: 'sorting'!

findFirstElementGreaterThanMiddleElement
  firstIndexGreaterThanMiddleElement := firstIndexGreaterThanMiddleElement
            + 1.
  [(data at: firstIndexGreaterThanMiddleElement) < middleElement]
      whileTrue:
          [firstIndexGreaterThanMiddleElement := firstIndexGreaterThanMiddleElement
                  + 1]!

findLastElementLessThanMiddleElement
  lastIndexLessThanMiddleElement := lastIndexLessThanMiddleElement - 1.
  [(data at: lastIndexLessThanMiddleElement) > middleElement] whileTrue:
          [lastIndexLessThanMiddleElement := lastIndexLessThanMiddleElement - 1]!

insertionSort
  startIndex to: stopIndex - 1 do: [:n | self insertSortedElementAt: n]!

insertSortedElementAt: anIndex
```

```
  | index element |
  element := data at: anIndex + 1.
  index := anIndex.
  [index >= startIndex and: [(data at: index) > element]] whileTrue:
          [data at: index + 1 put: (data at: index).
          index := index - 1].
  data at: index + 1 put: element!

medianOfThreeQuickSort
  self pickMedianElement.
  self partition.
  self recursivelySortPartitions!

partition
  firstIndexGreaterThanMiddleElement := startIndex.
  lastIndexLessThanMiddleElement := stopIndex.

  [self findLastElementLessThanMiddleElement.
  self findFirstElementGreaterThanMiddleElement.
  firstIndexGreaterThanMiddleElement <= lastIndexLessThanMiddleElement]
          whileTrue:
             [data swap: firstIndexGreaterThanMiddleElement
               with: lastIndexLessThanMiddleElement]!

pickMedianElement
  middleIndex := (startIndex + stopIndex) // 2.
  middleElement := data at: middleIndex.
  (data at: startIndex) > middleElement ifTrue: [self swap: startIndex].
  (data at: stopIndex) < middleElement
      ifTrue:
          [self swap: stopIndex.
          (data at: startIndex) > middleElement ifTrue: [self swap: startIndex]]!

recursivelySortFirstThenLastPartition
  startIndexStack
      add: firstIndexGreaterThanMiddleElement;
      add: startIndex.
  stopIndexStack
      add: stopIndex;
      add: lastIndexLessThanMiddleElement!

recursivelySortLastThenFirstPartition
  startIndexStack
      add: startIndex;
      add: firstIndexGreaterThanMiddleElement.
  stopIndexStack
      add: lastIndexLessThanMiddleElement;
      add: stopIndex!

recursivelySortPartitions
  lastIndexLessThanMiddleElement - startIndex
```

```
      > (stopIndex - firstIndexGreaterThanMiddleElement)
          ifTrue: [self recursivelySortLastThenFirstPartition]
          ifFalse: [self recursivelySortFirstThenLastPartition]!

sort

  [stopIndex - startIndex > 10
      ifTrue: [self medianOfThreeQuickSort]
      ifFalse: [self insertionSort].
  startIndexStack isEmpty not]
          whileTrue:
             [startIndex := startIndexStack removeLast.
             stopIndex := stopIndexStack removeLast]!

sort: array
  "Answer sorted data in ascending order
        using ACM #347 by Singleton."

  data := array.
  startIndex := 1.
  stopIndex := array size.
  startIndexStack := OrderedCollection new.
  stopIndexStack := OrderedCollection new.
  self sort.
  ^data!

swap: element
  "Swap data elements."

  data swap: middleIndex with: element.
  middleElement := data at: middleIndex! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

Sort class
  instanceVariableNames: ''!


!Sort class methodsFor: 'instance creation'!

array: anArray
  "Answer a sorted array."

  ^self new sort: anArray! !
```