Function Structure Example

Consider the following implementation for a RangeCombiner class.

- A "range" is two numbers "min" and "max" and it represents all the numbers from min to max.
- Our RangeCombiner class has methods for adding a new range: combiner.addRange(2.4, 3.5) will add a new range that represents all numbers from 2.4 to 3.5.
- The combiner is supposed to combine ranges that overlap. So for example if it already has the range from 2.4 to 3.5, and we add a range from 3.3 to 3.7, the two overlap and we should combine them to one range, from 2.4 to 3.7.
- Our implementation holds two arraylists, one for the minimums of the ranges and one for the maximums. They start out empty.
- Our implementation will hold these ranges in increasing order. For example the first range ends before the second range starts, the second range ends before the third range starts and so on. A method isRangeOrderValid() checks this property for us.
- When given a new range, our code searches through the list to find if this range overlaps with an existing range. If it does not, then it places the range at the end.

Group discussion

Look over the implementation in the handout[1], it should appear complicated.

1. What do you think are the characteristics of this code that make it hard to understand?
2. What could we do to make this class easier to understand? Work out some of the details of the change (are you introducing new methods, a new class, renaming things?).

Don't read further until you have thoroughly discussed the two questions above.

In order to address the problems, we are going to have a small refactoring session, and we will use the *incremental replacement* method that we saw earlier: Incrementally replace existing features with the new functionality. Recall the overall steps:

- Introduction:
  - Introduce a new class for the new structure, possibly via a refactoring step.
  - Introduce new fields to hold the new structure if needed.
- Scaffolding:

[1] activity3-1functionStructureHandout.html

  - Systematically update the new structure whereever the corresponding old structure elements are updated.
  - Systematically introduce extra parameters passing the new structure around along with the old structure.

- Teardown:

  - Systermatically replace accesses to the old structures with accesses to the new structure, until there are no accesses of the old structures.
  - Systematically eliminate no-longer-used parameters that refer to the old structures.
  - Systematically eliminate old structure updates.
  - Eliminate the old no-longer-used structures.

- Cleanup:

  - There are probably many places in the code where old structure values are needlessly produced, and we clean them up.
  - A number of methods probably need some refactoring, possibly moving to methods of the new class.

Here are the steps in detail:

- Introduction: We introduce a new Range class.

  - We start with a "Extract Method" on the entire body of the addRange method. We want to internally change the behavior of our methods but to maintain the same external interface. We will call the new method addRangeInternal, we'll worry about the name later.
  - Perform "Extract Parameter Object" on the new addRangeInternal method to a new inner class named Range with fields min and max.
  - We "Extract Method" on the first conditional, range.getMax() < range.getMin(), into an isEmpty method, which we then move to the range instance and inline its getMax() and getMin() calls.
  - We now create a new ArrayList instance named ranges to hold the ranges we are working with. This will slowly supplant the mins and maxs lists.

- Scaffolding: We gradually also add to ranges wherever we add to mins and maxs.

  - In the second conditional, which adds a range if it should go to the beginning, we add a ranges.add(0, range) before the return.
  - We also add a ranges.add(range) at the end of addRangeInternal method, where we are supposed to append the range to the end.
  - We now look at the call to insertValueAtIndexAndFixForward. Right before it, we need to create a newRange entry by putting together the newMin and newMax: Range newRange = new Range(newMin, newMax);. Later on we'll make a nice function for merging two ranges instead of having to compute the ends points ourselves.

- We add a new currRange parameter to insertValueAtIndexAndFixForward and pass the newRange value in at the call. We now proceed to look at the insertValueAtIndexAndFixForward method.
- We add ranges.set(i, currRange); to the initial part of insertValueAtIndexAndFixForward.
- Within the if conditional, we must also change the value of currRange to a newly created new range with the currMax and currMin values, and set it with ranges.set(i, currRange);. We also must remove the i+1 entry from ranges when we do so from min and max.

- Teardown: We gradually remove dependence on mins and maxs.
  - Our first step is the rangesOverlap method. It should be comparing ranges, not mins and maxes. We start at addRangeInternal and replace the mins.get(i), maxs.get(i) arguments with ranges.get(i).getMin(), ranges.get(i).getMax().
  - In the next call to rangesOverlap, we first add right before it a new Range nextRange = ranges.get(i+1); line. we replace the arguments to rangesOverlap(currMin, currMax, nextMin, nextMax) with uses of currRange.getMin()/getMax() and nextRange.getMin()/getMax().
  - We now want to make rangesOverlap use the ranges as parameters. To do that, we pick one of the uses, and perform "Extract Method" to a new method called rangesOverlap again, with parameter names range1 and range2. We tell the system to replace both instances.
  - Next we inline the old rangesOverlap method, we move it to be an instance of range1 and change its name to overlapsWith, change the remaining parameter's name to range2, and inline the various getMin and getMax uses. The method could still use some work, maybe we'll return to it later.
  - Now it is time to eliminate the mins and maxs uses. We do this as follows: Find a mins.get(i) use, then extract method on it call it tempMin for now, then change the body of tempMin from return mins.get(i); to return ranges.get(i).getMin(); then inline the tempMin method back and remove it. Repeat for max.
  - The only remaining issue is the usage of mins.size(). We can replace those with ranges.size().
  - We now need to eliminate the methods that were changing the mins and maxs lists. We have three kinds of methods: two kinds of add, a set, and a remove. We start by eliminating the set calls to both mins and maxs.
  - Next we eliminate the remove calls. Then the add calls.
  - Now we can remove the mins and maxs variables.
  - We still need to eliminate some parameters that are no longer really used. We look at insertValueAtIndexAndFixForward, and the currMin and currMax parameters there. We would like to eliminate them in favor of currRange. We look at where they are used, and find that their values get updated in the conditional. We change that update from currMin = Math.min(currMin, nextMin); to currMin = Math.min(currRange.getMin(), nextMin); and similarly for max.
  - Now we convert the currMin and currMax parameters to local variables, and further we inline their one use.

- It seems nextMin and nextMax are only used to construct the new currRange, so we replace their usage there with nextRange.getMin() and nextRange.getMax() respectively and eliminate them.
- Back in addRangeInternal we extract a local variable currRange for ranges.get(i), then inline the newMin and newMax uses.

- Cleanup: We simplify the resulting code to better use the new structures.
  - We examine the code for instances where min and max concepts are needlessly created. One part that remains messy is the creation of the merge of two ranges. Right now it is an extremely long call to the Range constructor with suitably computed endpoints. We extract this constructor call to a new method mergeRanges and replace both occurences. We then move the new method to be an instance method of range1, rename it to mergedWith, rename its remaining parameter to range, and inline the getMin and getMax calls.
  - We find a single use of the newRange local variable in addRangeInternal and decide to inline it.
  - Looking at the beginning of addRangeInternal, there is a call range.getMax() < ranges.get(0).getMin(). This seems to compare ranges to see if one precedes the other. We first create a local variable other out of ranges.get(0), then extract a method precedes out of range.getMax() < other.getMin(), move it to an instance of its first argument, change the remaining parameter to range and inline the getMax and getMin calls, then inline the local variable we had created.
  - Searching for other mentions of min and max outside of the Range class, we see the isRangeOrderValid method. The conditional seems to check exactly whether the range at index i does not precede the range at index i+1, so we replace that conditional with !ranges.get(i).precedes(ranges.get(i+1)).
  - Lastly, in printRanges we find the use String.format("%.2f--%.2f", ranges.get(i).getMin(), ranges.get(i).getM and extract it to a method of Range, after creating a temporary local variable for ranges.get(i). We further replace the loop in printRanges with a foreach loop.
  - Now we eliminate the getMin and getMax method altogether and feel better about the fact that the rest of the application does not need to know about min and max.

- Now for some more high-level cleanup.
  - We start with the overlapsWith method. Thinking about it, two ranges will overlap as long as one does not precede the other, so we can simply write the test as: !precedes(range) && !follows(range); where follows is a new method that is somewhat symmetric to precedes.
  - Next we look at the while loop in insertValueAtIndexAndFixForward. The nextRange variable is used in two places but it is a simple list lookup, so we inline it. TODO