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 needs to find where to put it. It searches through the list to find where this range fits, and merges it if needed with any ranges it overlaps. It goes as follows:
  - We loop over the ranges we have available.
  - At each index we check whether this is the right spot for our range. There are two reasons why that might be the case:
    1. The max of our range does not exceed the minimum at the current index, so our range should not go later.
    2. The minimum of our range does not exceed the current maximum, so again our range should not go later.

    In both cases, we insert our range in the current index, shifting everything to the right, and then start a merging process from the current index. Then we return.
  - Otherwise we continue with our loop until we’ve exhausted all ranges. In that case, our new range is at the end, past all the others, and we simply insert it there.

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?

[1] activity3-1functionStructureHandout.html

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: 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:
  - Systematically update the new structure wherever 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 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.
  - 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 want to pass range in to insertValueAtIndexAndFixForward instead of the range's min and max. In order to do that, we perform an "Extract Method" on the method call insertValueAtIndexAndFixForward(range.getMin(), range.getMax(), i); in order to get a method also called insertValueAtIndexAndFixForward but with parameters range and i. We then inline the old method.
  - The first four lines of the new insertValueAtIndexAndFixForward add to the i-th index of mins and maxs. We add ranges.add(i, range); there as well.
  - Within the if conditional, we must also change the value of range to a newly created new range with the currMin and currMax values, and set it with ranges.set(i, range);. 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 getters. We find a mins.get(i) use and extract a tempMins method from it. We change the body of this new method from return mins.get(i); to return ranges.get(i).getMin(); then inline the function tempMins back. We repeat the same trick for maxs.get(i).
  - Next we would like to deal with the rangesOverlap method. It should be taking two ranges as input instead of four values. We start by going to the call, and replacing currMin with range.getMin() and currMax with range.getMax().
  - Next, we extract a new nextRange variable from the expression ranges.get(i+1) used to create the nextMin and nextMax variables, and we inline and eliminate the nextMin and nextMax variables.
  - Now we are ready for rangesOverlap to use the ranges as parameters. To do that, we look at its use, and perform "Extract Method" to a new method called rangesOverlap again, with parameter names range1 and range2.
  - Next we inline the old rangesOverlap method, we move the new method 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 remaining mins and maxs uses. We already eliminated the getters. The only remaining issue is the usage of mins.size(). We can replace those with ranges.size(), by extracting a getSize method from them all, then changing it and inlining it back.
  - 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 variables that are no longer really used. We look at insertValueAtIndexAndFixForward, and the currMin and currMax local variables there. We would like to eliminate them in favor of range. 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, nextRange.getMin()); to currMin = Math.min(range.getMin(), nextRange.getMin()); and similarly for max.
  - Then we inline their one use. The new Range call seems aweful long now, we'll fix it.

- 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. We then move the new method to be an instance method of its first parameter, rename it to mergedWith, rename its remaining parameter to range.
  - Looking at the beginning of addRangeInternal, there is a call range.getMax() < range.getMin(). We "Extract Method" on it into an isEmpty method, which we then move to the range instance.
  - Next we see the comparison range.getMax() < ranges.get(i).getMin(). This seems to compare ranges to see if one precedes the other. We first create a local variable other out of ranges.get(i), then extract a method precedes out of range.getMax() < other.getMin(), move it to an instance of its first argument, and change the remaining parameter to range. We also notice the other part, range.getMin() <= other.getMax(), and extract it into a method doesNotFollow and move it around similarly. Then we inline the local variable other 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+1 does not follow the range at index i, so we replace that conditional with ranges.get(i+1).doesNotFollow(ranges.get(i)).
  - 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 inline and eliminate the getMin and getMax methods 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 they don't follow each other, so we can simply write the test as: this.doesNotFollow(range) && range.doesNotFollow(this);

- 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.
- Now it would be nice if the while loop no longer had to worry about the range variable: Once we insert the value in the i-th index, we should be able to use ranges.get(i) instead. In order to do that, let's see what we do with range: We merge it with the next range, then put the result into the i-th index. So if we replace the usage of rangein the conditional check with ranges.get(i), and we replace its usage in ranges.set(i, range) with ranges.set(i, ranges.get(i).mergedWith(ranges.get(i + 1)));, it will all still work. Now we can remove the line that was reassigning to the range variable.
- Now the while loop can be extracted into a method fixForwardFromIndex, and the old insertValueAtIndexAndFixForward can be inlined and removed.
- Looking at the while loop in fixForwardFromIndex, we can see that instead of breaking out of the else case, we can add the test in the conditional as part of the while loop's condition. Then we don't need the if conditional at all.
- We could go on with some more cosmetic refactorings, but the main part of the rewrite is now completed.