

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 handout¹, 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:
 - Systematically 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 (REMEMBER to CHECK your tests after each step):

- Introduction: We will introduce a new `Range` class.
 - We want to internally change the behavior of the `addRange` method without changing the external behavior of it. For this reason we want to create a new method. So select the contents (body) of `addRange` method, and perform “Extract Method” to obtain a new method called `addRangeInternal`. Not a great name but we'll worry about the name later.

- Now look at this `addRangeInternal` method. We will use it to create the new `Range` class. It has parameters `min` and `max` now. Perform “Extract Parameter Object” to create a new class using both parameters. You will need to specify that it is an *Inner class*. Call it `Range`.
- Next find the lines at the top of the file where the `mins` and `maxs` arraylists are created. Add a line to create and a new `ArrayList` instance named `ranges` to hold the ranges we are working with. Make sure to specify its type correctly so it will be able to contain `Range` instances.
- We will slowly supplant the usages of `mins` and `maxs` with usages of `ranges`.
- Scaffolding: We will gradually update the `ranges` list whenever we update the `mins` and `maxs` lists.
 - At the end of the `addRangeInternal` method we are performing adds on the `mins` and `maxs` lists, for the case where we add the new range at the end. Put after those a line that does `ranges.add(range);`, so that we also update the `ranges` list.
 - We turn our attention to the `insertValueAtIndexAndFixForward` method. We want to get to a point where this method takes as input a range instead of `min` and `max`. This is a 2-step process:
 - * Look at the *call* of `insertValueAtIndexAndFixForward`, namely `insertValueAtIndexAndFixForward(range, i)` and perform “Extract Method” to a method also called `insertValueAtIndexAndFixForward`. It will have a `range` parameter and an integer `i` parameter.
 - * Find the *definition* of the *old* `insertValueAtIndexAndFixForward` method, which took `min` and `max` as inputs, and “Inline” that method.
 - Now we get back to our task, which is to update the range values whenever corresponding `min` and `max` values change. Look at the top of the `insertValueAtIndexAndFixForward` method. You will see that we add `currMin` to the `mins` list and we add `currMax` to the `maxs` list. Right below those add instructions, insert a `ranges.add(i, range);` line to also update the `ranges` list.
 - Look at the body of the `if` conditional in `insertValueAtIndexAndFixForward`:
 - * We are computing new values for `currMin` and `currMax`. Right after those two computations, set `range` to equal a call `new Range(currMin, currMax)`. Thus when `currMin` and `currMax` changed, we accordingly changed `range` to agree with them. The next thing that happens in the body of that conditional is that we set some values in `mins` and `maxs`. We must also set a value in `ranges` when that happens, so add a line `ranges.set(i, range);`.
 - * The last line inside the `if` removes some entries from `mins` and `maxs`. We must remove a corresponding entry from `ranges`, so add a `ranges.remove(i);` call at the end.
- Teardown: We will gradually remove dependence on `mins` and `maxs` in favor of `ranges`.
 - Our first step is the getters. Our goal is to replace any mention of `mins.get(...)` with `ranges.get(...).getMin()`. We will effect this change in three steps:
 - * Find a `mins.get(i)` usage, and extract a `tempMins` method from it. When the option pops up, tell it to “Keep the original signature”, then tell it to substitute all *all* occurrences.

- * Now that the occurrences of `mins.get` only happen via the `tempMins` method, go in the body of this `tempMins` method and change it from `return mins.get(i);` to `return ranges.get(i).getMin();`.
- * Lastly, inline and remove the function `tempMins`. It has served its purpose.
- * Repeat the same technique for `maxs.get(i)`.
- * Lastly, use the same technique to `mins.getSize()`, and replace it with `ranges.getSize()`.
- We now want to gradually restrict references to `min` and `max` within the `insertValueAtIndexAndFixForward` method. We start with the `nextMin` and `nextMax` methods.
 - * Select the `ranges.get(i + 1)` section in one of them and perform “Extract Variable” on it to a new variable called `nextRange`. Replace *both* occurrences.
 - * Now inline and remove both `nextMin` and `nextMax`. They should have had two occurrences each.
- Our next goal is to reduce the current code’s dependence on `currMin` and `currMax`: These should both be obtainable from `range`, via `range.getMin()` and `range.getMax()` instead. As this will slightly change the semantics of the code, the system will not do this automatically for us, we’ll have to do it manually, and we’ll do it one step at a time.
 - * Start by replacing the `currMin` and `currMax` in the call to `rangesOverlap` with `range.getMin()` and `range.getMax()` respectively, and check your tests.
 - * Do the same with the `currMin` inside the `Math.min` and the `currMax` instead the `Math.max`, and check your tests.
 - * The next line defines `range` as `new Range(currMin, currMax)`. The values for those need to be the ones from the lines above, so replace the `currMin` in the constructor call with its value, `Math.min(range.getMin(), nextRange.getMin())`. Do the same for the `currMax`, then check your tests.
 - * The line below sets the values in `mins` and `maxs` using `currMin` and `currMax`. Now that our `range` variable has updated values, replace the `currMin` in that `mins.set(i, currMin)` with `range.getMin()` and similarly for `currMax`, then check your tests.
 - * At this point you should see two earlier settings of `currMin` and `currMax` be grayed out. Use the Alt-Enter intention menu to “Remove Redundant Assignment” on those two lines.
 - * Now go to the beginning of the `insertValueAtIndexAndFixForward` method. You should now be able to inline the `currMin` and `currMax` local variables there, with one occurrence each.
 - * We have now completely eliminated the mentions of `currMin` and `currMax` from this method; everything goes through the various `ranges` now.
- Next we would like to deal with the `rangesOverlap` method. Find where it is being used, and perform “Extract Method” on it to obtain a new `rangesOverlap` method that takes two `ranges` as parameters, instead of four doubles. Then inline and remove the old method, with the four parameters.
- Now move the `rangesOverlap` method so that it is a method of its first parameter (probably called `range`), then rename the remaining parameter to `range` and change the method name to `overlapsWith`.

- This new `overlapsWith` method has these two strange local variables, `min1` and `min2`. Inline them both. The resulting expression is fairly long, don't worry about it yet. We'll clean it up.
- 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`. Make sure that you do NOT delete one of the `ranges` operations as you do these steps.
 - * Start by eliminating the `set` calls to both `mins` and `maxs`, and check your tests.
 - * Next eliminate the `remove` calls. Then the `add` calls. Check your tests.
 - * Now the `mins` and `maxs` variable declarations near the top should appear grayed out. Perform the "Safe Delete" refactoring on them.
- Cleanup: Now we will 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, which happens within the conditional inside `insertValueAtIndexAndFixForward`. Right now it is an extremely long call to the `Range` constructor with suitably computed endpoints.
 - * Extract this whole `new Range(...)`; call to a new method called `mergeRanges`.
 - * Then we move this new `mergeRanges` method to be an instance of its first parameter (probably called `range`), we then rename the remaining parameter to `range`, and we rename the method to be called `mergedWith`.
 - Looking at the beginning of `addRangeInternal`, there is a call `range.getMax() < range.getMin()`. Perform "Extract Method" on it into an `isEmpty` method. Make sure to tell it to "Keep the original signature". Then it move to an instance method of its parameter, `range`.
 - Next we see the comparison `range.getMin() <= ranges.get(i).getMax()`. This seems to compare ranges to see if one precedes the other. We'll extract it to a method but we need to prepare it first:
 - * Extract a local variable `other` out of `ranges.get(i)`. This is a temporary extraction to get the right parameters to our methods.
 - * Extract a method `doesNotFollow` out of `range.getMin() <= other.getMax()` (keep the original signature).
 - * Move the new method `doesNotFollow` to be an instance of its first argument, and change the remaining parameter to `range`.
 - * Now inline the local variable `other` that we 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 replace that conditional with `ranges.get(i+1).doesNotFollow(ranges.get(i))`, and check your tests.
 - Lastly, in `printRanges` we find the use `String.format("%.2f—%.2f", ranges.get(i).getMin(), ranges.get(i).getM`
 - * Create a temporary local variable `range` out of `ranges.get(i)` (replace both occurrences).

- * Extract the whole `String.format(...)` expression to a method `getSimpleFormat` with parameter `range`, then move it to be an instance method of `range`.
 - * Back in the `printRange` method, perform the “Replace with foreach” intention to replace it with a simpler for-each 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 `range` in 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.