Activity 2-2 Introduction to refactoring

- Refactoring is a change in the structure of the code, without changing the behavior of the code.
- Its intent is primarily to make the code:
  - easier to read and understand
  - more amenable to future changes and the addition of new features

Refactoring can therefore be thought of as serving two main uses:

1. Cleaning up the code to make it more friendly to future "visitors". We will call this *the boyscout rule*: **Try to leave the code a little better than you found it**.
2. Preparing the code for the addition of some future elements. We must be careful with this second one, and not do it willy-nilly without a concrete idea of what new elements we want to implement. Trying to anticipate all future requirements and preparing the code for them results in overly convoluted code that is hard to work with; and it turns out at the end of the day that most of the features you thought would be added were in fact not added. This situation is typically described by the acronym YAGNI[1] for "you aren't gonna need it". The key underlying goal should always be to *keep the design simple*.

Refactoring takes many forms, and a more precise list of these refactorings is described on this page[2] as well as the refactoring book[3].

As an example of this process, consider the extremely straightforward but quite long-winded solution to the grade-processing activity on the first page of the handout[4].

Group activity: Discuss what features of this code make it hard to read, and possibly hard to change in the future. Do this before reading on.

So let's list a number of problems here:

- The variable names are really not very descriptive at all. Most of them are single letters, and they don't make it clear what is going on. So we will want to rename them. **Renaming** is probably the simplest of all the refactorings.
- There is a large switch statement, whose goal seems to be to figure out what numeric value should correspond to a particular letter grade. This part has a very clear logic of its own, so it should probably be a separate function. **Extracting** a piece of code to create a new method or a new local variable is another fairly simple but extremely important form of refactoring.

[1] https://martinfowler.com/bliki/Yagni.html
[2] ../cheatsheets/refactorings.html
[3] https://learning.oreilly.com/library/view/refactoring-improving-the/9780134757681/
[4] activity2-2refactoringHandout.html

- There are various bits and pieces of computations and calls to the scanner class that are by themselves somewhat obscure. We could use a comment for them, or we can also extract them into methods and use the method names to describe their intent.
- Thinking about it, we possibly want to have a separate class that represents a grade, and it knows about the corresponding value for each letter, which letters count for credit, etc.

Phase 1: Renaming and method extraction

We can start with some simple changes:

- Rename the variable t to total. We need to do this consistently, and also to make sure there isn't already a variable named total. A good rule of thumb if you do it manually is this: Change its name in its declaration, then find all the places that the compiler complains about. Note how many places this had to change in, phew lots of work!
- Rename the variable c to courses. This counts the number of courses that we count towards the gpa.
- It seems that l stands for a letter grade so we'll use that name instead. We do this using the automated refactoring menu.
- We now select that whole switch statement, and extract it into a separate method. Since the switch statement changes the total value, we will need to make it return an int. And really if we notice the various statements like t += 1.33; we probably want our function to simply return the 1.33, and do the addition at the end. So our code will say something like: total += getGradeForLetter(letter);. We do these changes gradually: First we create the new method, copy the code over and fix any syntax errors. Then we replace the original code, and make sure the tests still pass.
- The statement !letter.equals("W") is really meant to determine if the letter grade should count for credit. We therefore want to replace it with a method call: countsForCredit(letter)
- It seems that the line courses == 0 ? 0 : total / courses computes the total gpa, so we'll extract it into a function computeGPA.
- There are some scanner.next("\\s*\\w+\\s*"); lines at the beginning. Actually these two lines serve different purposes: The first reads the course department prefix, while the other reads the course number and letter grade. We extract them in two function readPrefix and readCourseNo. In the future, the code for these might no longer be the same.
- The scanner.next("[ABCDFW][+−]?"); pattern reads a letter grade. We again extract it to a readLetterGrade method.
- The conditional that follows, if (scanner.hasNewLine()) ... basically reads to the end of the current line. We should extract this to a method as well: readToEndOfLine().

- Lastly, the String.format("Courses: %d\nGPA: %.2f\n", courses, gpa); phrase should probably also be its own method, called formatSummary.

At this point our code looks as in the second page of the handout. Notice how much more readable the main function's operations are.

Phase 2: Class extraction, moving elements around

Looking at all these, a couple of things stand out that suggest creating some new classes:

- It seems quantities like courses, total and gpa are often used together. It would make sense for them to all be part of the same class. In fact, since gpa is simply computed from the other two, it would make sense for such a class to basically maintain the courses and total, but to be able to report the gpa when asked. Perhaps such a class could be called Summary.
- It seems clear that the logic for computing the numeric equivalent of a letter grade, and whether that grade should count for credit or not, is very separate from the rest, and perhaps should be its own Grade class.
- Finally, a number of methods deal with processing the input using the scanner. These should also be in a dedicated class, perhaps called a GradeReader.

Let's see how we might go about creating these classes.

- We start with the Summary class. We will start by performing "Extract Parameter Object" on the computeGPA method, to turn those two parameters into one summary object of a new Summary inner class of Main.
- This left a new Summary(courses, total) argument inside the computeGPA method. We extract it to a variable summary.
- We still have the local variables courses and total around. We gradually replace them with summary.courses and summary.total after we move the creation of summary to just before the while loop. In the process, we have to tell the system that the two fields should not be final.
- We now inline the two local variables, which now are only used by the Summary constructor.
- In the Summary class, we inline the two parameters in the constructor, as their value should really always start at 0. We then move the initializations to the declarations, and delete the now empty constructor (it will use the default constructor).
- The computeGPA method should really be an instance method of the Summary class, we move it there and inline its use of getCourses and getTotal.
- We similarly want to move the formatSummary method. But we cannot yet until it has the summary as it parameter. We start by inlining the gpa local variable.

- Now we perform "Extract Method" on the expression formatSummary(summary.courses, summary.comput to get format method with summary as its parameter. Then we inline the old formatSummary method.
- Then we move the format method to an instance method of the Summary class, and inline any getters used in its body.
- Now we are left with the four lines that update the courses and total values. Thinking about it more, these should happen in a single update, something like summary.add(units, points). Or even better, just summary.addGrade(letter). This letter will probably become the grade later on. So we extract a method from those lines and then move it to the summary class.
- Lastly, up to now the Summary class was an inner class of Main. We now move it to its own file, which means we have to make a number of our methods public (or at least package-protected).

Our code now looks like the third page of the handout.

Now we proceed with our second class extraction. It looks like it might be nice to have the concept of a *grade* as more than a single letter, namely an entity that has some functions. Maybe later we can turn it into an enum, but for now it would be good to simply have a Grade class.

- We start from the addGrade method in Summary, which currently takes as input a letter. We perform "Extract Parameter Object" on it to turn that letter into a Grade class as an inner class of Summary.
- We now have to lines in addGrade from which we can extract new methods of the Grade class: Main.getGradeForLetter(grade.getLetter()); should become a getPoints method and then moved to the grade instance. And Main.countsForCredit(grade.getLetter()) should be extracted to a countsForCredit method and then moved to the Grade class.
- We now look at the bodies of these two methods and inline the two Main... methods that are no longer needed.
- We also move Grade to the upper level and adjust the access modifiers of some of its methods.
- Lastly, we notice that getLetter is used only internally in Grade, and we inline it.

We will also handle the processing steps. We effectively want to replace all uses of the scanner with uses of a newly-created Processor, with the scanner as part of its constructor. In order to achieve this, we'll have to perform a step that by itself does not appear useful, namely we'll extract the whole while loop into a single new method. This is a temporary step so that we have a place where we can perform "Extract object".

- We perform "Extract Method" on the while loop to obtain a processAll method.
- We perform "Extract Parameter Object" on just the scanner parameter of the processAll method. We then perform "Inline" on the processAll method to eliminate it.

- We now need to perform "Extract Method" to all the various lines that use the processor, as they should mostly be methods there. We extract method called hasNext, then move it to be an instance method of processor.
- We repeat this process for readPrefix(processor.getScanner()) to a method called readPrefix, for readCourseNo(processor.getScanner()); to a method called readCourseNo, for readLetterGrade(processor.getScanner()); to readLetterGrade and finally for readToEndOfLine(processor.getScanne to readToEndOfLine. We convert each to an instance method of Processor. We then inline the Main... bodies of these new Processor methods, to remove the original methods that are still in the Main class.
- We then inline all the uses of getScanner and eliminate the method from Processor. The rest of the world does not need access to our scanner.
- Lastly, we look back at our main while loop. It seems that there are at least four lines there that concern how the processor will process a line, in terms of the order in which events will happen. That really ought to be in the processor. So we will move it there. The conversion of a letter to a Grade should also be a part of that, but right now it is intertwined with the addGrade call. We start by extracting a grade variable from the new Grade(letter); expression. Then we grab all but the last line of the body of the while loop, and perform extract method for it to a method getNext which we then convert to an instance method of the processor. We then inline the grade variable we temporarily created.
- Finally, we pull this Processor class to the upper level.

And now we see the final form of our code, nicely separated into four classes. Notice how simple the processGrades method has become:

- It creates a summary and a processor.
- It keeps reading grades using the processor, and adding them to the summary.
- It then returns the summary using its formatter.