**Question 1**

To begin, we should define what we are measuring when we talk about the quality of code. From class and from the Clean Code textbook, we learned that quality code should be easy to read, easy to debug, and easy to maintain. We can examine whether or not an arbitrary piece of code is readable, easy to debug, and easy to maintain by examining it at different granularities. We can look at single lines of code in isolation, single methods in isolation, or whole classes. At each of these levels, we can tell whether or not the code is readable, easy to debug, and easy to maintain, and therefore come up with some measure of its quality.

First, we can examine individual lines of code and determine their quality. For a line to be easy to read, debug, and maintain, it should probably only do one thing. At my old job as a sound designer for video slot machines, where the sound designers were responsible for writing XML implementations of their sounds in the game, one or two of my colleagues would write lines of code that looked like this:

<Playlist> <PlaylistID> WinSound </PlaylistID> <CommandList> <SetVolume> 75 </SetVolume> <Pan> 0 </Pan> <Play><Sound> rockin\_win\_tune.wav </sound> <LoopCount> 1 </LoopCount> </Play></CommandList> </Playlist>

This line does a bunch of different things: it creates a playlist, names it, sets its volume, sets its panning, plays a sound, sets a loop parameter for the playback. Because this line has so much in it, it’s hard to pick out a single value of a single element. If something in the game sounded weird, and all 1000 Playlists were one liners like this in the XML document, it would take us a very long time to fix it. Let’s change the code to have one element per line like this:

<Playlist>

<PlaylistID> WinSound </PlaylistID>

<CommandList>

<SetVolume> 75 </SetVolume>

<Pan> 0 </Pan>

<Play>

<Sound> rockin\_win\_tune.wav </Sound>

<LoopCount> 1 </LoopCount>

</Play>

</CommandList>

</Playlist>

This is clearly much easier to read, and it’s because each line only does one thing. Its readability leads to it being easier to debug and maintain, and is therefore higher quality.

We can also determine the quality of a method by reading it. According to the Clean Code textbook, there are several things one can look for when determining the quality of a method. First, we can simply look at how long it is. If it’s really long, it’s going to be less readable, and therefore lower quality. Second, we should be able to determine what a method does just by reading its name. To achieve this, according to the textbook, methods need to be named well and methods should only do what they say they do in their name. In an extreme example, a method with the signature setField(Obj val) shouldn’t also return something. If it did, debugging code that used this method would become very difficult. If we come across a method that is doing more than its name suggests, we know that the code is of lower quality.

Finally, we can determine the quality of a class by reading its code. According to the Clean Code textbook, one way to determine the quality of a class is to determine whether or not it follows the Single Responsibility Principle. Just like for methods and for single lines of code, it basically means that classes should only be responsible for one thing. If a class is responsible for too much, code becomes difficult to follow and maintain. On page 138 of the textbook in Listing 10-2, there’s an example of a class called SuperDashboard that accesses UI components and tracks version numbers. Let’s say we needed to update the way that versioning is tracked. It would not be obvious that SuperDashboard would be the class to go to in order to do this. Therefore, this class has code that is difficult to maintain, and is of lower quality than if it were broken into two classes.

Another way we can read a class and determine its quality is by looking for dependencies to other classes, also known as coupling. To show how coupling can make code difficult to debug, consider this toy example: We have a class called Line that creates lines of some integer length. Then we have a class named Square with a Line as an instance variable and an area method that looks like this:

public int getArea(){

return line.getLength() \* line.getLength();

}

If later we decide that Lines should have a length that is a double instead of an integer, we also will have to change the getArea() function in the Square class in order to prevent returning the wrong area. Any other method that uses Line’s getLength() method will also have to change. Now our code is more difficult to maintain, since a change to one class requires that we change another. Code in which changing one class will not break another will have less coupling, be easier to maintain, and therefore be of higher quality.

To determine the quality of code, we must determine if it is readable, easy to debug, and easy to maintain. We can make this determination by examining the code at three different levels. First, we can examine individual lines and determine if they are readable. Second, we can examine individual methods and ensure that they are not too long and that they only do the one thing that they are named to do and nothing more. Finally, we can examine classes and ensure that they are not too coupled to other ones and that they follow the Single Responsibility Principle. Once we have examined the code in this way, we can have some measure of its quality.

**Question 2**

To answer this question, we should continue with the definition of quality that we had in the answer to the first: quality code is easy to read, easy to debug, and easy to maintain. We should also assume that refactoring is like a proofread. After we write a first draft of our code to pass a test or tests, we look over it and find better ways to abstract code into fields, methods, and classes so as to remove repetition and make it more maintainable. With these definitions in mind, I argue that there is a fairly strong correlation between refactoring and code quality – an 8 on a 1 to 10 scale. I say an 8 because occasionally I believe that refactoring can be taken a little too far. We will examine three cases, two in which refactoring improved the quality of the code, and one in which it didn’t.

The first case involves an Android application that I recently finished writing that keeps track of how many bikes are available at Divvy stations around the city (inspired by Professor Cohen’s Ruby homework in his Web Development class). In the application, the user is presented with a list of all of the stations to choose from, and clicking on them pops up a dialog that shows the clicked station’s current details. The user can also add a station to her favorites, which are contained in another list. Clicking on a station in the favorites list pops up a dialog to show its details. When I began programming the app, I started with the class that dealt with the functionality of the main screen that showed the list of all of the stations. I put the code for showing the details dialog directly into the class. Next, I worked on the class that contained the favorites list. When I got to coding the dialog in the favorites, I realized that I needed the exact same code that I had previously coded for the main screen’s dialog. Rather than copy and paste, I decided to refactor. I extracted the code for the dialog out of the main screen class and put it in its own dialog class. The constructor for the dialog class took a station name, and it handled everything else. My main screen class now only needed to call a method to construct the dialog and another to show it. Not only did this refactor shorten my classes and make them more readable, but it also made my code more easily maintained and expanded. Let’s say I wanted to add more information to the dialog. I would only need to change the dialog class. Let’s say I wanted to add another screen that leveraged that same dialog. I’d just need to call the dialog constructor method and show method in the new screen class. Because the refactor has led to more maintainable code, it is responsible for an improvement in quality.

The second case involves the test class that I wrote for the bus ticket program in this course. One of the requirements of the program was that a ticket to ride on a certain day cannot be sold if that day is in the past. The first day I worked on the test class, I wrote a bunch of tests in which I hardcoded ride dates, and all of my tests passed. A week later when I ran the tests to continue working, the first thing I realized was that most of my tests were failing. It was the hard coded dates! Ride dates that were not in the past a week earlier were in the past now. I needed to refactor. I wrote a method to generate a weekday date and one to generate a weekend date that would always be valid based on the current date. I replaced all of my hardcoded dates with these methods. This was obviously a huge improvement in quality. Now the tests could be run on any day in the future without having to rewrite anything.

The third case is one in which code was abstracted too much (likely due to refactoring) so that the maintainability, and therefore quality, of the code was actually worse. It again involves my job as a sound designer for video slot machines. In every slot game that we made, there was a win animation where some numbers on the screen would sequentially count up how much money the player won while music played. The music in every game was almost always the same, and if it wasn’t, it was just a short loop that would end when the money stopped counting up. One day, a game designer asked me to write custom win music for a game. I did, and when the programmer put the music in the game, we saw that the numbers stopped counting up before the music ended. I asked the programmer if he could change the length of the numbers counting to match the music, and he said that he couldn’t. I escalated the request to his boss, and he told me that he didn’t know how to change the timing either, but would try to figure it out with me. After drilling down through several levels of abstraction, we found ourselves in a class that had the lengths of the count up animations. This class was part of the platform package, one that was used as the foundation for every game that they programmed. It turns out that several years earlier, when we were using the same win music for every game, the programmers had decided that it was redundant to code in the count up timings for every game. They refactored, and in doing so, abstracted the count up timings into a class. That class eventually got pulled into the platform package so the engineers could forget about it, which they had. In this case, the refactor had made the code less maintainable. Something as simple as changing the timing of an animation took two programmers half of a workday, when it should have taken one engineer just a few key strokes. I note that such a specific case like is probably much rarer than the other cases I described.

In the cases I described above, we see that refactoring code can usually bring about more quality. Our first attempt to write often involves hard coding things that we will probably want to reuse later, and abstracting those things into classes and methods during a refactor often makes it easier to maintain. However, sometimes after multiple refactors, we will abstract something so deeply that we don’t know where to go to change it when we need to. This is likely rarer of an event, which is why I’ve given an 8 out of 10 for how correlated refactoring is to code quality.