Stephen Smith
CMSI 402
Professor Johnson
2-25-15

Assignment #3

Problem 9.1

Postfactoring is a way of removing the “bad smells” from the code. It is what slows code decay and makes it possible to perform future changes. In postfactoring, many of the illogically named pieces of code are fixed to fit its purpose, long functions are split of into smaller functions, and larger classes are broken down into smaller ones. This makes the code more readable, adaptable, and accommodating to current or future programmers of the code.

Problem 9.3

1) Prefactoring - moving a function from one class to another is like prepping it before making another change to the program that will use that function
2) Postfactoring - extracting a super class is like cleaning up code to help future readers understand the relationships of classes better
3) Postfactoring - same reason as 2, extracting a component class will typically reduce the amount of code and clean it up
4) Prefactoring - merging classes is not really done as cleanup but more as prep work for whatever comes after to interact with the new class

Problem 9.4

```
void printPosition() {
   int i, j;
   char text[1024] = "1234567890";
   int text_length = 10;
   char array_to_searchl[4] = "23";
   int array_to_search1_length = 2;

   cout << getPositionOfTextInString(text, text_length,
                                     array_to_search,
                                     array_to_search1_length);
}

int getPositionOfTextInString (char[] text,
                               int textLength,
                               char[] arrayToSearch,
                               int arrayToSearchLength) {
   int position1 = -1;
   for( i = 0; i < text_length - array_to_search1_length + 1; i++ ) {
      bool found = true;
      for( j = 0; j < array_to_search1_length; j++ )
         if( text[i+j] != array_to_search1[j] )
            found = false;
      if( found ) {
         return i;
      } else {
         return -1;
      }
   }
}
```

Problem 9.5

```
public class A {
  public static void main( String args[] ) {
    double c = Double.parseDouble( args[0] );
    double t = c;
    double EPSILON = 1e-15;

    if (c >= 0) {
      if (c == 0) {
        c = -c;
      }
      while( Math.abs( t - c/t ) > t * EPSILON ) {
        t = (c/t + t) / 2.0;
      }
      System.out.println( t );
    } else {
      System.out.println(<<Error: the number is smaller than 0>>);
    }
  }
}
```

We only need to know whether c is greater than or equal to 0, and in the case that it is equal to 0 we negate it. If it is less than 0 then we want to print out that it is an erroneous input. Other than that, the rest of the refactoring was code duplication.

Problem 15.2

Insufficient knowledge leads to code decay because a project can have such a large domain that little of that knowledge base makes it into useful documentation. You are then stuck with programmer(s) writing code for something they do not fully understand. A lack of meaningful identifiers also contributes to code decay. This makes it harder to perform concept location in order to find and/or modify current code.

Problem 15.4

I would have a function that took in the year as a parameter and determined whether it was a leap year or not. Inside that function I would call the function mentioned in the question.

Problem 15.7

Homogenous software is software in which all of its modules are in the same stage of the software life span. Heterogeneous software is software whose code consists of some modules that are evolvable, other decayed modules that cannot be evolved, or stabilized modules that do not need to be evolved.

Problem 15.11

Reengineering is the process of reversing code decay by reorganizing and/or rewriting the decayed code of the whole system or substantial parts of it. Reverse engineering is the first phase of reengineering. It is in this phase that programmers analyze the old code and extract the relevant information from it.

Problem 12.3

1) Being rewarded based on LOC(lines of code)/day would have the advantage of pushing programmers to do more work in one day; however, the drawback would be that it is hard to measure, especially when it comes to programming styles and the various types of languages out there. Rewarding programmers in this way could lead to less effective code and more “filler” code in order to give the impression that a programmer is contributing a lot to a project.
2) Duplicating code only increases the size of the project (not that we’re fighting for memory that much anymore), and it clutters the focal point. If another programmer were to search the source code for a specific spot surround by Bob’s duplicated code, it might slow his/her understanding of what Bob is trying to say.
3) The manager should employ the use of code inspection. By inspecting the code, the manager will see the duplication of Bob’s work. If the bug is within the duplicated code, then the manager can tell Bob to refactor it into one consolidated block of code to isolate and fix the bug.

Problem 12.8

A defect log is a list of all the known defects in the software. It is an important measurement of quality and lets the programmers know where the defect is occurring as well as any accompanying information. This way programmers can attempt to avoid or solve the bug when writing or modifying existing code that involves the defect.

Problem 13.4

It is advantageous to separate programmers from testers not only because each group requires different skills, but also to avoid a potential conflict of interest. Developers might try to hide the shortcomings of their code by creating tests that fit the ideal scenarios. In this case, the ideal scenarios will be tested but not the outlying scenarios rendering tests that are not exhaustive.

Problem 13.10

Extreme programming resembles that of an AIP, or Agile Iterative Process. It aims for a simplistic design and planning on the part of the programmers who meet fairly often to discuss the accomplishments and shortcomings of the day. It cranks out more releases and better code on the scale of AIP. The DIP, or Directed Iterative Process, seems to involve managers and higher ups in the company. This slows down the evolution process which is not what extreme programming is meant to do.