FIT4004 - System Validation & Verification, Quality and Standards

Supplementary Assessment Part II Metrics Essay

Student Name: Wanyu Yin
Student ID: 24141232

Introduction

The purpose of writing this essay is to demonstrate the understanding after reading the paper of two metrics assessing coupling by Bieman and Kang, as well as being able to provide the description of the calculations of the two metrics using my own code example, and the comparison between the two metrics and LCOM metrics that was taught during FIT4004 lectures.

What is more, there will be the explanation based on my own opinion whether the two metrics introduced in the paper are of great use to measure the cohesion.

Concepts of Cohesion

Generally speaking, cohesion means that the level of relations among the components inside one module within a system. The higher the cohesion is, the more reliable and the more reusable the module is, and vice versa. With high cohesion, each component inside the module will be very hard to separate and distinguish apart.

The paper mainly discusses about the class cohesion. For a class, the components are the instance variables and the methods defined within the class and inherited from the parent class. The relations can be described as follows:

- If a method use one instance variable, they are related.
- If two methods use the same instance variable, the two methods are related.

Cohesion is often described with another concept called coupling. The definitions of the two concepts share some similar ideas, but they are two different things. Cohesion focuses on only one module, while coupling measures the relationship between two modules.

Calculations of The Two Metrics

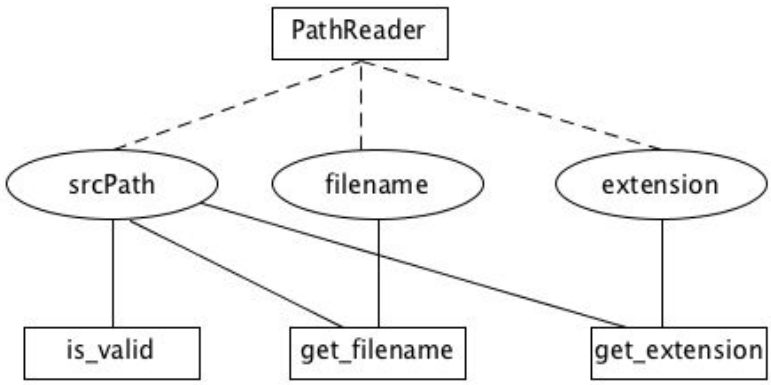
The two metrics introduced in the paper are *Tight Class Cohession (TCC)* and *Loose Class Cohesion (LCC)*.

I believe the best way to show how to calculate the two metrics is by using my own code. Here is the example code written in Python:

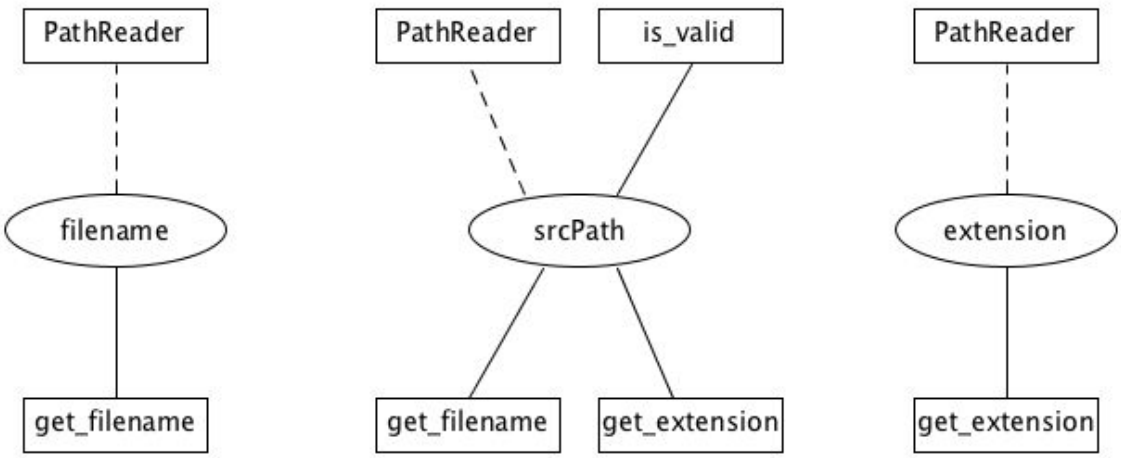
```
class PathReader:
  def __init__(self, srcPath):
    self.srcPath = srcPath
    self.filename = ''
    self.extension = ''

  def get_filename(self):
    if self.split_srcPath():
      str_list = self.split_srcPath()
```

```
        for i in range(len(str_list)-1):
            self.filename = self.filename + str_list[i] + '.'
        if self.filename[-1] == '.':
            self.filename = self.filename[:-1]
        return self.filename

    def get_extension(self):
        if self.split_srcPath():
            str_list = self.split_srcPath()
            self.extension = str_list[-1]
        return self.extension

    def split_srcPath(self):
        if self.srcPath is None or len(self.srcPath)==0 or len(self.srcPath.split('.'))<2:
            Print 'Invalid source path…'
            return False
        else:
            return self.srcPath.split('.')
```

The following two graphs are showing the relations for the components of the class PathReader (for better reading, "self" is removed from both the instance variables and the methods):



MIV relations



Connections for each instance variable

Since there is no parent class for PathReader, *abstracted class* is equal to *local abstracted class*, that is, AC(C) = LAC(C). Thus, the calculations are as follows:

TCC (relative number of directly connected methods):
AC(C) = LAC(C) = [[{srcPath, filename}, {srcPath, extension}, {srcPath}]]

Number of direct connections in AC(C) (share one or more common instance variables between two methods):

NDC(C) = 3

N methods in the class:

N = 3

Maximum possible number of direct or indirect connections in the class:

NP(C) = N * (N - 1) / 2
= 3 * (3 - 1) / 2
= 3

TCC(C) = NDC(C) / NP(C)
= 3 / 3
= 1

LCC (relative number of directly or indirectly connected methods):
Number of indirect connections in AC(C):

NIC(C) = 0

LCC(C) = (NDC(C) + NIC(C)) / NP(C) = (3 + 0) / 3 = 3/3 = 1

Therefore, by using the above calculation sequences, TCC and LCC can be calculated, and the corresponding answers are both equals to 1.

Comparison with LCOM Metric

Lack of Cohesion in Methods (LCOM) metric shares one similar part with TCC and LCC. They both have a set that describes each pair of methods shares and implements at least one same instance variable within one class.

However, the calculations are obviously not the same. TCC measures the percentage of direct connectivities in the maximum possible method pairs, while LCC takes the indirect connectivities into account. LCOM focuses on the variant between the direct connectivities and indirect or no connectivities.

The higher the TCC is, the higher the cohesion of the class is. Same for LCC. if these two metrics are high, it indicates that the class has more methods sharing the same instance variables directly or indirectly. But for LCOM, the higher, the less cohesive the module will be, which means the less methods directly use the same instance variables.

Unfortunately, as shown clearly above, LCOM cannot measures the indirectly connectivities. This might cause the analysis of the module not accurate enough, because indirectly calling is very common in method writing. Therefore, the loosely connected module analysed by LCOM can actually turn out to be of high cohesion if it uses lots of indirectly connectivities. In further, from the paper written by Bieman and Kang, they expressed the ideas that LCOM is not very suitable for finding partial cohesive classes, that is, indirect connectivities between methods.

How Well The Two Metrics Measure Cohesion

From my point of view, I reckon that TCC and LCC are quite useful in analysing the cohesion. By measuring the cohesion, the developer can find out which modules might not be well designed by targeting at the modules with lower TCC and LCC.

In my example code, the class is quite simple and only contains three instance variables and three methods. Since all the methods in the class share the same instance variable "self.srcPath", the results of TCC and LCC are both 1, which indicate that my class has very high cohesion. My original design purpose is to create a small and highly cohesive class, and the results of TCC and LCC prove my idea.

According to the paper, TCC and LCC can not only measure the methods defined in the class itself but also measure the inherited methods. But in their experiment results, the measurements of the classes that are heavily reused via inheritance show less cohesion. The class from my code example does not inherit from a parent class, so I could not really how well this part could work.

Conclusion

In conclusion, TCC and LCC metrics are useful in measuring the classes which do not heavily depend on reusing methods via inheritance. I will highly consider utilising TCC and LCC metrics in the cohesion measurement of small code modules. However, for large project which contains lots of inheritance reuse, the measurement might not be very effective.