Comparing the Results of Relation Analysis and Coupling Metrics

Initial Case Study

Jonne Itkonen

Department of Mathematical Information Technology,
P.O. Box 35, FIN-40014 University of Jyväskylä, Finland
ji@mit.jyu.fi

Abstract. Relation Analysis, finding simultaneously changing software modules, is a powerful way to find out unspecified logical couplings during software development. Dependency metrics like abstractness and instability of packages calculated from the number of abstract, afferent and efferent classes in a package are used to qualify the object-oriented design of software. This study tries to find out can Relation Analysis and dependency metrics strengthen each other. Some minor but promising results are achieved, suggesting that detailed research on analysis of the now calculated metrics in time should be considered.

1 Introduction

Relation Analysis (RA), as part of the QCR-method developed by Gall et al. [1], identifies parts of software that contain unspecified logical couplings. These logical couplings might be impossible to notice using traditional dependency analysis techniques, as they usually do not manifest themselves as code level relations, but as relations in developers mind. An example might be that if module A is changed with respect of operation O, module C should be changed too.

The idea of this study is to compare the results of RA to the results obtained from calculating coupling metrics between the modules. It is not expected that a strong correspondence is found between RA and coupling metrics, but a positive result would be, if RA could ensure the results of coupling metrics. Of course, the coupling metrics ensuring the results of RA would be a nice finding also, but that might just say that the couplings were specified. Either way, the developers of the inspected software are interviewed to get their opinion on the results.

2 How things are done

RA is based on finding changes made to software at the same time, i.e. within a small time period. Changes to a group of modules within this small time period are considered happened together, and recorded as an event consisting of a set of module identifiers and a timestamp. The reoccurring of this event

is calculated through the evolution of software. Should modules that have no planned dependencies change together repeatedly, they are considered having an unspecified logical coupling.

The recording of changes is done with a widely used source code management system, the Concurrent Versioning System (CVS). It has a logging ability that shows what files have changes, when, how, and by whom. CVS does not record changes atomically, that is, it doesn't apply the changes of the changed files to the code repository as one entity, as modern source code management systems do. Nevertheless, even on modern systems this grouping by time might be necessary, as time period used can be a couple of minutes long and so contain a few atomic transactions.

For coupling metrics, those presented by Robert C. Martin in [2] are used. These metrics count the number of all classes N_c and abstract classes N_a, with the number of afferent C_a and efferent C_e classes in a package. Afferent classes are those that don't belong to the current package P, but do depend on the classes inside the package P. Efferent classes are those that are inside package P and depend on classes outside package P.

From these metrics we then calculate the instability $I = \frac{C_e}{C_a+C_e}$ and abstractness $A = \frac{N_a}{N_c}$ of the package, and distance from main sequence $D = \frac{|A+I-1|}{\sqrt{2}}$. Instability has a range of $[0, 1]$, wher $I = 0$ means maximally stable package and $I = 1$ means that the package is maximally instable. Abstractness has also the same range. Packages that have $A = 0$ have no abstract classes where packages with $A = 1$ have nothing but abstract classes.

The meaning of main sequence is clearly shown if we plot the abstractness and instability values on a union square like in Figure 1. Martin [2] defines the area near (0,0) as *the zone of pain* as these packages are rigid and hard to change, should the need arise, but many packages are dependant of these. Packages that RA shows to be under heavy change should not be located here. The area near (1,1) is *the zone of uselessness*, as packages here are most abstract and most instable, but useless, because not many packages depend on these. Packages should be near the main sequence, the diagonal $y = 1 - x$. Packages away from this, as denoted by distance from main sequence D should be investigated further.

RA is done by self-made software that extracts the data from the log of CVS. This data is then visualized using self-made script that outputs a graphics file rendered with GraphViz. As RA-software does currently not collect all the needed data, CVSAnalysis is used to get the missing data, namely number of lines, lines added and deleted, and the creation date and last modification date for a class.

In calculating the dependency metrics Dependency Finder was used. The data it produced is changed to a GNUPlot-renderable file by a self-made XSLT-transformation and a Python-script. As can be seen from the number of different techniques and tools used, some integration and reengineering is needed to clarify the data acquisition process.

3 Preliminary Results

The software under inspection is a study management system called Korppi developed at the University of Jyväskylä. More details of Korppi and applying RA to it is given in [3]. RA was done with 90-second timeframe. One noticeable candidate for unspecified logical coupling is between class `kiurubeans.SearchFreeSpaceForm` and class `kolibribeans.AppointmentForm2` with weight of 17 (that means these classes have changed 17 times at the same time-period) and another is one with a weight of 11 between class `kiurubeans.SearchFreeSpaceForm` and class `kolibribeans.Appointment`. These couplings seem to be quite meaningless, as couplings with weight over 100 exist. There are total of 768 logical couplings between `kiurubeans` and `kolibribeans` the sum of their weights being 1079 giving the average of 1.4. This value should be compared to the strongest coupling candidate that is between `kiurubeans` and `kotkabeans`, that has 5036 couplings with weight sum of 5932 giving the average of 1.8.

Table 1. History data for three classes

	Created	Modified	Revision	+	-	LOC
`AppointmentForm2`	2004-09-29	2005-05-06	81	7227	4606	783
`SearchFreeSpaceForm`	2004-09-29	2005-04-22	64	5748	3580	829
`Appointment`	2004-03-12	2005-05-10	93	2716	597	56

Nevertheless, there seems to be nothing wrong here. Looking at the names of these classes and their ages it can be assumed that they were developed together. Interview with the developers confirmed this. Ages and data on changes for these classes is given in Table 1. Fields are name of the class, creation date, modification date, number of revisions, lines added, lines deleted and lines of code including empty and comment lines.

The dependency metrics for the packages of these classes are given in Table 2. All three packages are located quite near the main sequence, but as the distance metrics (D' and D) shows, `kolibribeans` might be a bit suspicious.

Figure 1 is a graphical representation of the data in Table 2. As the picture shows that the classes of this project do not locate near and along the main sequence. Most are instable and concrete. The developers, however, know this. They are not writing this software as object-oriented as possible. Figure 1 shows also, that the two zones of problematic packages are quite empty. The few packages that are located in these zones are `kotkabeans` and `tools`. RA shows no strong logical couplings between these packages.

There is a group of Sync4J-related packages in the software that also lie near the problematic zones. RA shows a logical couplings with an average weight of 1,4 between `kotkabeans` and `sync`. This is a promising finding for which not enough time has been given yet.

Table 2. Dependency Metrics for packages `Kolibribeans` and `Kiurubeans`

	C_a	C_e	I	N_c	N_a	A	D'	D
`kolibribeans`	28	112	0.8	19	0	0	-0.2	0.14
`kiurubeans`	19	141	0.88	81	8	0.1	-0.02	0.014
`kakibeans`	15	68	0.82	31	6	0.19	0.01	0.007
`kakibeans.ability`	3	27	0.9	8	2	0.25	0.15	0.11
`kotkabeans`	313	220	0.41	164	8	0.05	-0.54	0.381
`tools`	162	45	0.22	14	6	0	-0.78	0.553

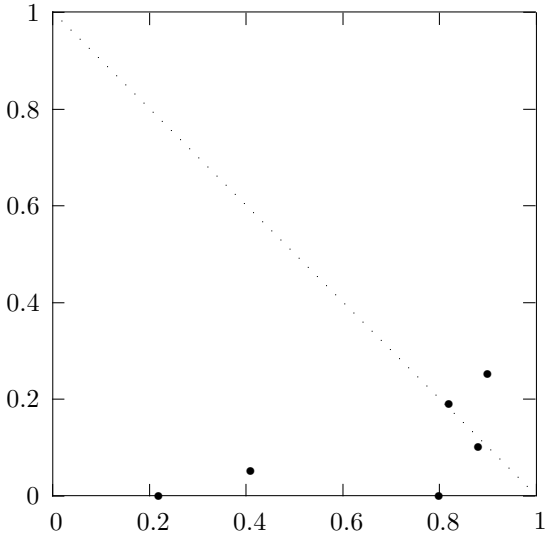

Fig. 1. $A - I$ Graph for packages from Table 2

It is interesting to note that packages marked by RA to contain possible unspecified logical couplings and being developed strongly are also indicated such by the distance from main sequence metric D.

4 Conclusion

Based on this very restrictive study it seems like some correspondence between RA and the selected dependency metrics exist. This has to be assured by examining the data thoroughly and including the research of how the data changes over time to.

Also, the presumption that logical couplings do not manifest themselves in code might be hasty. Static code analysis, namely finding duplicate code in different modules, might discover some of the unspecified logical couplings. Nevertheless, RA could be used to qualify the existence of these couplings between code duplicates.

The results should also be compared against the results of qualitative code estimation methods.

References

1. Gall, H., Jazayeri, M., Krajewski, J.: Cvs release history data for detecting logical couplings. In: Sixth International Workshop on Principles of Software Evolution (IWPSE'03). (2003)
2. Martin, R.C.: Agile Software Development: principles, patterns and practices. Pearson Education, Inc. (2003)
3. Itkonen, J., Hillebrand, M., Lappalainen, V.: Application of relation analysis to a small java software. In: Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04), IEEE Computer Society (2004)