Code Decay Triangulation using Metrics, Experts' Opinion and Defect Counts

Juan Fernández-Ramil
Computing Dept. The Open University, UK
j.f.ramil@open.ac.uk

David Reed
IT Division of a UK Government Department
ydj_reed@yahoo.co.uk

10th edition of the BElgian-NEtherlands software eVOLution seminar BENEVOL, Brussels, 8-9 Sept 2011 http://soft.vub.ac.be/benevol2011/

Motivation

In contrast to physically engineered artefacts, software does not deteriorate through use. Code quality, however, may *decay* (i.e. deteriorate) through the process of software evolution (a.k.a. maintenance). Such decay may have negative human, technical and economic consequences. For example, software maintainers may find that the code is becoming excessively complex. Evolution may become more time consuming and difficult than it should. Other stakeholders may not receive the functional improvements they are waiting for in time. Unexpected side-effects may emerge when new changes are implemented. Defect fixing may get harder. And so on...

The problem of code decay (a.k.a. code aging, excessive complexity, 'spaghetti' code) has been identified and discussed a long time ago [e.g., Lehman 1974, Parnas 1994]. There are many code decay empirical studies in the literature [e.g. Eick et al 2001]. There are, at least, three different ways of trying to assess the level of code decay in a particular system: direct measuring of the code through software metrics, surveying experts' opinion about code quality and using indirect measures (e.g. process related measures such as defect counts). It isn't known whether these three different ways will converge to the same insights when applied to a particular system.

In this extended abstract, we briefly report the findings of a case study in which software metrics, a developers' questionnaire and defect counts were compared and used in an attempt to rank the software's components with respect to their level of decay. We aimed at achieving a greater clarity on the 'details' of how to measure code decay in a particular context. We also wanted to provide the organisation owing the software with a ranked list of components which could be use to prioritise any refactoring or replacement efforts.

The Case Study

The software used as a case study was a proprietary business critical information system. The system handled an important database which is used nationally by many stakeholders. Any errors in the system and in the database may have serious legal and financial implications. The system was initially implemented in 2004 following the PRINCE2 methodology and using mainly Borland Delphi, a variant of Object Pascal. At the time of the study the system had evolved for four years, with 19 releases. At the most recent release the system consisted of approximately 225,000 lines of Delphi code including comments. Further details can be found in [Reed 2009].

Data Collection

The study involved the collection of three types of data: code metrics, defect counts and expert opinion. The code metrics included McCabe complexity, coupling (CBO), afferent and efferent coupling and lack of cohesion (LCOM2). The expert opinions were gathered via a specially designed questionnaire. The number of reported defects was obtained from the documentation and manually assigned to each of the subsystems.

The metric data was visualised by plotting *point values* (average values) per release and box-plots (i.e. abstracted views of the distributions) for the first and most recent releases. From the box-plots, the *tail length* and the *tail volume* was calculated for each of the 11 subsystems. Changes in metrics values were measured relative to the first release rather than in absolute terms. A first version of the questionnaire was generated and sent to a small number of experts who could give comments on it. The questionnaire was then revised based on their feedback and then sent to the real developers. It was answered by 10 out of 12 developers. In order to normalise defect counts by the size of the system, the cumulative number of defects was divided by the current size of the system in number of lines of code.

Main Results

The three types of data provided some evidence that could be interpreted as decay being present. However, the convergence was not complete. For example, the average McCabe complexity increased slightly from 2.96 to 3.08 (4.1%) during the 4 years of evolution. Tail volumes increased for complexity, CBO and afferent coupling, showing evidence of code decay. Surprisingly, tail volumes decreased for efferent coupling and LCOM2, showing improvement rather than decay. Six developers said that the system has become more complex; three developers indicated that the complexity has stayed the same and one developer said that the system has become less complex. Cumulative defect values (normalised by size) showed a positive slope (increasing trend) from month 22.

Seven ranking pairs (based on point values, tail length, tail volume, questionnaire and defects) for the 11 subsystems were compared using Kendall's and Spearman's rank correlation measures. The results ranged widely from positive correlations (e.g. Spearman's Rho value of 0.7 for the pair 'point values – defects' to negative correlations (e.g. Kendall's Tau value of -0.4 for the pair 'tail volume – questionnaire').

Despite the evidence showing, overall, that the code has decayed, it was found that different types of measurement may lead to different results. Code decay is multi-dimensional. Careful examination is needed to interpret which measures are more meaningful in a given context. In general, expert opinion seems to be the most reliable source of information, followed by code metrics (at distribution level) and finally defect counts. Defect counts can vary widely due to, for example, changes in the testing effort, without necessarily indicating code decay. Within code metrics, the analysis based on distributions (box-plots) was found to be more insightful than point values (averages). The latter generally 'compress' the tail of the distribution where the most complex code elements reside and in this way may hide the parts of the code where the actual code decay is actually happening.

Conclusion

Code decay symptoms are not easy to triangulate, that is, to confirm (or not) through different types of measurement whether the code has suffered from quality deterioration. In this case study an initial approach based on code metrics, questionnaire and defect counts showed mixed results. For example, some metrics showed deterioration while others showed improvement. Moreover, subsystem decay rankings of possible decay based on different types of information are not always leading to the same results. Despite all this, the methodological approach used in this case study could be used by a software organisation to start an internal discussion and reflection on the evolutionary 'trajectory' of the system and on the possible measures to improve code's quality where it is most needed. How to apply code decay measurement approaches in a given context or project is not immediately clear and needs experimentation. All this, makes code decay detection a difficult problem for practitioners and an interesting area of research which combines the software evolution and the software measurement topics.

References

[Eick et al 2001] S.G. Eick et al, Does Code Decay? Assessing the Evidence from Change Management Data, IEEE TSE, 27(1), pp. 1-12 , 2001.

[Lehman 1974] M.M. Lehman, *Programs, Cities and Students – Limits to Growth?,* Inaugural lecture, Imperial College of Science, Technology, London, 14th May 1974

[Parnas 1994] D.L. Parnas, Software Aging, Proc 16th ICSE, Sorrento Italy, pp 279-287

[Reed 2009] D. Reed, *Code Decay – Examining Evidence from Expert Subjective Assessment and Metrics*, M801 Master's Dissertation, Computing Dept., The Open University, Milton Keynes, U.K., March 2009.