Related works

It proved difficult to find related works which also consider the correlation between quality metrics and code volatility. However, we mention the following papers which describe pieces similar to which we used in our approach.

Munson and Elbaum looked at comparing the complexity of sequential builds in order to measure the impact of code change regarding fault proneness [6]. They aimed at determining a suitable fault surrogate (which they dubbed code churn), and furthermore established that code churn is associated with measures of program quality. This research has some similar themes compared to our research in that they investigate how churn correlates with quality (faults); however, they don’t delve into the concept of occurrences of churn across multiple releases.

Graves et al. posit that *“In general, process measures based on the change history are more useful in predicting fault rates than product metrics of the code: For instance, the number of times code has been changed is a better indication of how many faults it will contain than is its length.”* in their work [7]. Here we clearly observe similar themes compared to our proposal of code volatility. Their model considers collections of files as an individual module, compared to our view that individual classes are the core building blocks. One of the arguments they make regarding standard complexity metrics is that they correlate too highly with lines of code: “…*numbers of lines of code in modules are not helpful in predicting numbers of future faults once one has taken into account numbers of times modules have been changed.*” This work has similar findings to our research, most notably that using the sum of contributions from all changes in a module’s history can effectively predict fault potential. They proceed one step further and also consider a module’s age in the prediction calculation, which is something we did not do.

One other work which looks at code change and quality is Nagappan and Ball’s research [8]. In it they describe two measures (M5 and M6) which take into account "the cumulative time that a file was opened for editing from the VCS". While this isn't exactly our definition of volatility, it does represent something similar. This concept was used to cross-check the other measures described in their research and it is interesting to our paper since it introduces the concept of quantifying volatility, albeit differently (total time open vs. # of changes across different releases).

A number of previous studies have been done on large OSS projects such as jEdit, Eclipse, Apache Ant, and Mozilla Firefox to detect the coupling between different java files, classes and methods in order to build prediction models. Ying et al. developed an approach which is based on association rule mining to find recommendations on potential file changes. For example if a developer changes a particular file, then her approach will recommend the other files that will need to be changed. Their findings were superior to ours mostly due to a richer set of data and by omitting confidence.

Similarly, Zimmermann et al. developed a tool named ROSE that works on association rule mining of CVS [10]. Their main concern was to find finer-grained entities, which was absent in Ying’s work. They used association rule mining with minimum support and confidence. Their tool seems to be powerful in terms of suggesting and predicting the next likely code elements to change. It also prohibits and warns about incomplete changes. We took inspiration from both these works and built association rules in a similar fashion. We found that we were able to perform some predictions on future change; however, again their results were superior to ours for various reasons such as richer data set, and using a weighted set of rules (top 10).

[1] H.M. Olague, L.H. Etzkorn, S. Gholston and S. Quattlebaum, "Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Process," IEEE Trans. Software Eng., vol. 33, no. 6, pp. 402-419, June 2007

[2] R. Marinescu, "Detection Strategies: Metrics-Based Rules for Detecting Design Flaws," Proc. 20th IEEE Int',l Conf. Software Maintenance, 2004

[3] A. Trifu and R. Marinescu. Diagnosing design problems in object oriented systems. In Proceedings of the Working Conference on Reverse Engineering (WCRE), Pittsburgh USA, 2005 [4] J. Bansiya and C. Davis, “A Hierarchical Model for Object-Oriented Design Quality Assessment,” IEEE Trans. Software Eng., vol. 28, no. 1, pp. 4-17, Jan. 2002.

[5] S. Chidamber and C. Kemerer, “A Metrics Suite for Object-Oriented Design,” IEEE Trans. Software Eng., vol. 20, no. 6, pp. 476-493, June 1994.

[6] Munson, J. C., Elbaum, S., "Code Chum: A Measure for Estimating the Impact of Code Change," Proceedings of IEEE International Conference on Software Maintenance, 1998, pp. 24-31.

[7] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy, “Predicting Fault Incidence Using Software Change History,” IEEE Trans. Software Eng., vol. 26, no. 7, July 2000.

[8] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density." in Proceedings of the International Conference on Software Engineering (ICSE 2005), St. Louis, Missouri, USA, 2005, pp. 284-292.

[9] A.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carroll, “Predicting Source Code Changes by Mining Change History,” IEEE Trans. Software Eng., vol. 30, no. 9, pp. 574-586, Sept. 2004.

[10] T. Zimmermann, Weißgerber, P., Diehl, S., Zeller, A., “Mining Version Histories to Guide Software Changes”, IEEE Transactions in Software Engineering, 31(6), pp. 429-445, 2005.