Feature Volatility Assessment

Warren Baelen, Yuanfang Cai
Department of Computer Science
Drexel University
{st95b589, yfcai}@cs.drexel.edu

Robert Dyer, Hridesh Rajan
Department of Computer Science
Iowa State University
{rdyer, hridesh}@iastate.edu

I. INTRODUCTION

With the development of new modularization techniques, such as aspect-oriented programming and feature-oriented programming, assessing and comparing their differences in different circumstances becomes important for the user to chose, compare and synthesize these techniques. Numerous studies have been conducted to assess and compare different software modularization techniques in terms of their impact on software modularity and stability when new features are added [3][4][5].

At the same time, it has been realized that traditional software metrics based on coupling and cohesion measurement are not effective in terms of measuring key properties of modularity, such as separation of concerns and option value generation. Researchers have proposed and applied new assessment techniques. Concern-based metrics, such as Concern Diffusion over Components (CDC), Concern Diffusion over Operations (CDO), Concern Diffusion over Lines of Code (CDLOC), have been used to compare aspect-oriented vs. object-oriented implementations[3][4][6]. Baldwin and Clark's net option value analysis and design structure matrix modeling [1] have been used to assess how well different paradigms can effectively generate option values [2][6][8].

These studies show that the impact of a modularization technique not only depends on the techniques itself, but also depends on the design and nature of a particular feature. For example, some modularization techniques make crosscutting features more stable, such as exception handling, while making other types of features more volatile. It is also possible that these modularization techniques do not make much difference for certain types of features. The concern-based metrics only count the number of components, operation, and lines of code that are influenced by the feature, but do not directly assess how stable these components are. The option-based metrics measure the effects of overall design, blurring the impact on each feature.

In the paper, we propose a feature stability/volatility measurement to explicitly show which modularization technique is the best for which type of features, and to show how the stability of each feature changes over time. Using these feature stability measurements, the designer can not only compare and contrast different modularization techniques, but also track which features is most volatile or how maintenance activities change feature volatility over time. For a highly volatile feature, the components implementing it will be changed frequently.

Our idea is to combine concern diffusion measurement with the internal coupling of components. The rationale is that: feature stability depends on the stability of the components that implements the feature, and the stability of the feature is thus the summation of the *stability* of its components. As a result, we first need to measure the stability of each component.

Traditionally, the stability of a software component is measured by the number of dependents (fan-out dependencies) divided by the total number of dependencies (both fan-in and fan-out). That is, the more dependents a component has and the fewer it depends on, the more stable a component is. However, it is possible that all the other components a component depends on are stable in the sense that they do not subject to any environmental changes.

Based on the assumption that environmental factors, such as features, are the drivers of software changes, our recent work [6] proposed a new software volatility measure: the more features influence a components (*EnvrImpact*), and the more dependents it has (*ImpactScope*), the more volatile it is. That is, for component *i*, $Volatilty_i = EnvrImpact_i * ImpactScope_i$. Accordingly, the volatility of a feature is the summation volatility of each component implementing it: $FeatureVolatilty = \Sigma Volatility_i$.

II. MEASURING MOBILEMEDIA FEATURE STABILITY

To preliminarily assess the effectiveness of this feature volatility measure, we applied it to different types of features of MobileMedia, a software product line [3][4][6][7][8] system that is widely studied. The MobileMedia system has eight releases, each adding a new feature to the previous release. These new features are categorized into three types, each of which has dramatically different characteristics: (1) alternative features, such as

functions related to photo, music or video; (2) optional features, such as selecting favorite media; (3) mandatory features, such as exception handling.

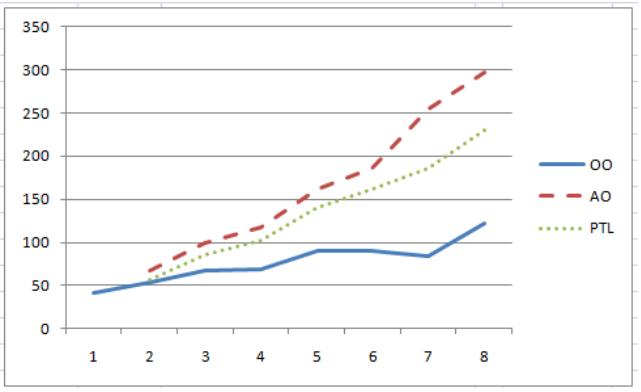

Figure 1. The Create Photo Feature Volatility

We calculated the volatility of each feature in each version, designed in Java, AspectJ, and Ptolemy [7] respectively, to understand which design is best for which feature. Figure 1 shows the volatility of the create photo feature from release 1 to 8. The chart shows that using OO design will make the feature most stable, and using AO design will make this feature most volatile. The data suggests that using OO to implement this feature is the best choice.

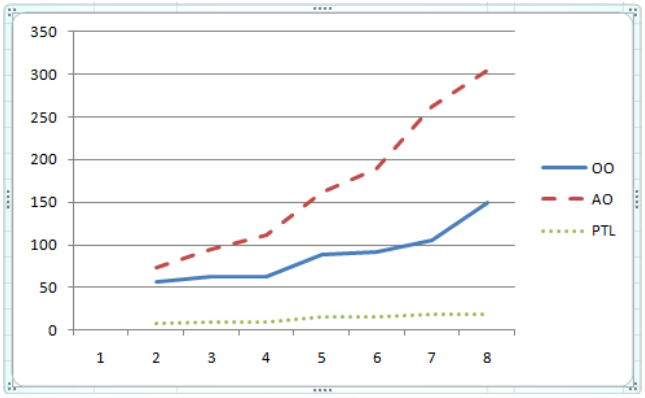

Figure 1. The Exception Handling Feature Volatility

Figure 2 shows the volatility value of the exception handling feature. The data shows that Ptolemy is the best choice for this feature because the resulting design is far more stable than the other two choices. The AO design appears to be the worse choice, which has been confirmed by previous studies.

We also measured the volatility of optional features, such as setting favorite photos or music. The results depend on the concrete feature, that is, not all the optional features were impacted in the same way by these different paradigms. For example, for the favorite photo feature, both Ptolemy and aspect-oriented design appear to be most stable with 0 volatility. This means that the components that implement this feature do not influence other components, and that these two paradigms appear to have the same ability in terms of accommodating this feature.

REFERENCES

[1] Carliss Y. Baldwin and Kim B. Clark. *Design Rules, Vol. 1: The Power of Modularity*. The MIT Press, 2000.

[2] Cai, Y., Huynh, S., and Xie, T. A framework and tool supports for testing modularity of software design. in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering* (*ASE 2007*), pages 441–444, November 2007.

[3] Figueiredo, E. et al.: Evolving software product lines with aspects: An empirical study on design stability. In *Proceedings of the 30th International Conference on Software Engineering*, Leipzig, Germany, 2008.

[4] Garcia, A. et al. Modularizing design patterns with aspects: a quantitative study. In *Proceedings of the 4st international conference on Aspect-Oriented Software Development (AOSD)*. 2005, pages 3–14.

[5] C. Sant'Anna et al, "On the modularity of software architectures: A concern-driven measurement framework," in *Proc of the 1st European Conference on Software Architecture (ECSA)*, Sep. 2007.

[6] K. Sethi, Y. Cai, S. Wong, A. Garcia, and C. Sant'Anna,"From retrospect to prospect: Assessing modularity and stability from software architecture," in *Proceedings of the Joint 8th Working IEEE/IFIP Conference on Software Architecture and 3rd European Conference on Software Architecture (WICSA/ECSA)* 2009.

[7] Hridesh Rajan and Gary T. Leavens, "Ptolemy: A Language with Quantified Typed, Events," ECOOP '08: *22nd European Conference on Object-Oriented Programming*, July 2008, Paphos, Cyprus.

[8] Robert Dyer, Mehdi Bagherzadeh, Hridesh Rajan and Yuanfang Cai, "*A Preliminary Study of Quantified, Typed Events*," the Empirical Evaluation of Software Composition Techniques (ESCOT 2010), Rennes and St. Malo, France, March 2010.