Towards Assessing the Impact of Crosscutting Concerns on Modularity

Marc Eaddy Alfred Aho

Department of Computer Science
Columbia University
New York, NY 10027
{eaddy,aho}@cs.columbia.edu

1. INTRODUCTION

The goal of aspect-oriented programming is to modularize crosscutting concerns. To fully appreciate this goal, we must first understand how crosscutting concerns affect modularity and software quality, and to what extent. This is hard to quantify, partly because terms such as "crosscutting", "concern", and "modularity" are ill-defined [11] [1], and partly because the scope of the crosscutting concern problem is unknown.

We propose a research agenda whose first step is to formalize the crosscutting concern problem. We present a set theoretic *concern model* (§2) that formalizes terminology and provides a foundation for a suite of *concern metrics* (§3) for quantifying the distribution and separation of concerns. Second, we must determine the extent of the problem. We advocate a *concern assignment methodology* (§4) whereby *all* the concerns of a program (and their associated code fragments) are rigorously identified. The third step is to argue convincingly that crosscutting concerns negatively impact modularity and software quality. For this, we propose to correlate our concern metrics with traditional modularity metrics and external quality indicators such as fault proneness [10] (§5).

2. CONCERN MODEL

For the concern model, we extend the work of Berg, Conejero, and Hernández [1], which we briefly summarize here, by applying a set theoretic treatment. This makes the model more amenable for defining our concern metrics.

The concern model consists of an abstract source domain *S*, a target domain *T*, and a *trace relation R*. The source domain is an abstract *concern domain*, the elements of which are individual concerns. An example of a well-defined concern domain is a formal requirements specification.

The target domain may be another concern domain, or an *implementation specification*, the elements of which are individual software components (e.g., files, classes, methods, statements). The target domain can also be treated as the source domain for another (S, T, R) tuple. This allows a concern to be traced from initial identification to subsequent phases of the software lifecycle [1].

Scattering can now be defined as the case when **a source element is related to multiple target elements**. ***Tangling*** is when **a target element is related to multiple source elements**. [1] These definitions concur with [6].

We define crosscutting as follows: a ***crosscutting concern*** **is a scattered concern, i.e., a concern related to multiple target elements**. This definition agrees with [7, p. 4]. Berg et al. [1] and some other researchers define crosscutting with respect to scattering *and* tangling; however, we do not believe tangling is an essential ingredient. We conjecture that tangling is included to indicate inherent complexity, the assumption being that a tangled concern is both harder to modularize and hurts modularity more than a nontangled concern. However, no evidence exists to support this claim.

3. CONCERN METRICS

We recast the *closeness metrics* created by Wong et al. [13] using our set theoretic concern model, to form the basis for our *concern metrics*. (We assume that all of the concerns of the program have been identified beforehand and associated with their corresponding program statements, perhaps using the methodology described in §4.)

3.1 Degree of Scattering (DOS)

Concentration (*CONC*) [13] measures how many of the statements related to a concern *s* are contained within a specific component *t* (e.g., a file, class, method):

$$CONC(s,t) = \frac{\begin{matrix} SLOCs\ in\ component\ t \\ related\ to\ concern\ s \end{matrix}}{SLOCs\ related\ to\ concern\ s}$$

The drawback of this metric is that it does not give a sense for how scattered a concern is and it does not allow concerns to be compared. To resolve this, we created the *degree of scattering* (*DOS*) metric (for brevity we do not show its derivation):

$$DOS(s) = 1 - \frac{|T| \sum_t^T \left(CONC(s,t) - {}^1\!/_{|T|}\right)^2}{|T| - 1}$$

where *T* is the set of program components. DOS is a measure of the variance of the concentration of a concern over all components with respect to the worse case (i.e., when the concern is equally scattered across all components). A concern that is completely localized has a DOS of 0, whereas a concern that is uniformly distributed has a DOS of 1.

A high DOS indicates the implementation of a concern is highly scattered (crosscutting). A localized implementation is a defining characteristic of a module, so a concern that is crosscutting is by definition not modular. Furthermore, the components across which the implementation of the concern is scattered are less modular than if the crosscutting concern were not present. Without providing an equation for measuring modularity, we can assume that it is inversely proportional to the average degree of scattering (ADOS, obtained by averaging DOS over all the concerns of the program).

3.2 Degree of Focus (DOF)

Dedication (DEDI) [13] measures how many of the statements contained within a component *t* are related to concern *s*.

$$DEDI(t,s) = \frac{\begin{array}{c}SLOCs\ in\ component\ t\\ related\ to\ concern\ s\end{array}}{SLOCs\ in\ component\ t}$$

Again, the drawback is that it is hard to get a sense for how well concerns are separated in a component. To resolve this, we created the *degree of focus* (*DOF*) metric:

$$DOF(t) = \frac{|S| \sum_s^S \left(DEDI(t,s) - {}^1\!/_{|S|}\right)^2}{|S| - 1}$$

where *S* is the set of concerns. DOF is a measure of the variance of the dedication of a component to every concern with respect to the worse case (i.e., when the component is equally dedicated to all concerns). DOF is 0 when a component's "attention" is uniformly divided among every concern, and 1 when a component is dedicated to one concern. The average degree of focus (ADOF) gives an overall indication for how well concerns are separated in the program.

DOS and DOF can also be used to evaluate a software design, guide refactoring decisions, and compare refactoring alternatives. They measure the relationship between logical entities (concerns) and physical entities (components). Hence, they provide more information than traditional metrics (e.g., the CK metrics), which only measure relations between physical entities. For example, degree of scattering provides a direct measure of the change impact associated with changing a requirement, and more accurately predicts change cost than the CK metrics.

Previous concern metrics ([14], [9], [11], and [6]) detect the *presence* of a concern, but do not measure the *degree* of presence. A concern whose implementation is split 99-1 between two components would be considered equal to a concern split 50-50. Thus, common refactorings such as consolidating redundant code into a reusable function would not be deemed beneficial by their metrics.

4. CONCERN ASSIGNMENT METHODOLOGY

Our concern metrics require that statements of the program are associated with concerns. To allow concerns and components to be compared and to determine the extent of the amount of crosscutting present in a program, a *complete concern assignment* must be performed.

Concern assignment is a hard problem [2]. Manual efforts to reverse engineer source code to divine the implemented concerns can lead to assignments that are inconsistent (different people have different assignments), inaccurate (a statement is assigned to the wrong concern), and incomplete (not all statements or concerns are assigned) [3] [11] [5] [8]. While more consistent, automated methods also lead to inaccuracies [5] [4].[1]

Our goal is to obtain a complete and accurate picture of the nature and scope of the crosscutting present in a program. To this end, we advocate using a *formal requirements specification* as the concern domain (as opposed to reverse engineering or mining the concerns). For concern assignment, we adopt a form of requirements tracing where we manually assign a requirement concern to a program statement ***if the removal of the requirement would necessitate the removal or modification of the statement***.

We call this a *removal dependency-based assignment*. In our experience, this methodology is more straightforward than the *minimal subsets, minimal increments*-based technique [3] because it eliminates the need to reverse engineer the concern domain. Assignment reduces to a simple litmus test to determine if a removal dependency exists between two well-defined elements.

We refer to the set of statements associated with the requirement as the *concern slice* for that concern. The concern slice directly indicates the change

[1] Nevertheless, we are considering incorporating some type of automation to ease the assignment burden.

impact associated with removing an existing requirement, and more importantly, it approximates the impact associated with modifying a requirement or adding a new requirement. More evidence is needed to confirm this hypothesis.

5. CONCLUSION

We established a formal foundation for assessing the crosscutting concern problem. We presented metrics that directly measure the distribution and separation of concerns in a program, and showed how they provide unique insights into modularity and change impact.

We outlined a methodology for identifying all the concerns of a program and associating them with every program statement, which allows us to obtain a complete picture of the crosscutting present in the program, and evidence of the scope of the crosscutting problem. For example, for one case study we observed that 53% of the feature concerns of the program where crosscutting at the file level, indicating a significant potential for improving the modularity of the program and motivating the need for techniques to modularize crosscutting concerns.

We plan to conduct a series of case studies to validate our metrics and methodology, and to correlate our metrics with other modularity measures [12] and external quality indicators such as fault proneness [10]. One study is almost complete and the preliminary results are very promising. For example, we now know an accurate concern assignment (as described in §4) is essential for ensuring measurement repeatability.

6. REFERENCES

[1] K. v. d. Berg, J. M. Conejero, and J. Hernández, "Analysis of Crosscutting across Software Development Phases based on Traceability," *Wkshp. on Aspect-Oriented Requirements Engineering and Architecture Design*, 2006.

[2] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, "The concept assignment problem in program understanding," *Intl. Conf. on Software Engineering*, 1993.

[3] L. Carver and W. G. Griswold, "Sorting out Concerns," *Wkshp. on Multi-Dimensional Separation of Concerns*, 1999.

[4] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwe, "Applying and Combining Three Different Aspect Mining Techniques," *Software Quality,* 14(3):2006.

[5] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Transactions on Software Engineering,* 29(210-224, March 2003.

[6] E. Figueiredo, A. Garcia, C. Sant'Anna, U. Kulesza, and C. Lucena, "Assessing Aspect-Oriented Artifacts: Towards a Tool-Supported Quantitative Method," *Wkshp. on Quantitative Approaches in OO Software Engineering*, 2005.

[7] R. E. Filman, T. Elrad, S. Clarke, and M. Aksit, *Aspect-Oriented Software Development*. Boston, MA: Addison-Wesley, 2005.

[8] A. Lai and G. C. Murphy, "The Structure of Features in Java Code: An Exploratory Investigation," *Wkshp. on Multi-Dimensional Separation of Concerns*, 1999.

[9] A. Lai and G. C. Murphy, "Capturing Concerns with Conceptual Modules," 2001.

[10] N. Nagappan, T. Ball, and A. Zeller, "Mining Metrics to Predict Component Failures," 2006.

[11] M. Revelle, T. Broadbent, and D. Coppit, "Understanding Concerns in Software: Insights Gained from Two Case Studies," *Intl. Wkshp. on Program Comprehension*, 2005.

[12] S. L. Tsang, S. Clarke, and E. Baniassad, "An Evaluation of Aspect-Oriented Programming for Java-based Real-time Systems Development," *Intl. Symp. on OO Real-Time Distributed Computing*, 2004.

[13] W. E. Wong, S. S. Gokhale, and J. R. Horgan, "Quantifying the closeness between program components and features," *Journal of Systems and Software,* 54(2):87-98, 2000.

[14] C. Zhang and H.-A. Jacobsen, "Quantifying Aspects in Middleware Platforms," *Aspect-Oriented Software Development*, 2003.