Identifying, Assigning, and Quantifying Crosscutting Concerns

Marc Eaddy, Alfred Aho
Department of Computer Science
Columbia University
{eaddy,aho}@cs.columbia.edu

Gail C. Murphy
Department of Computer Science
University of British Columbia
murphy@cs.ubc.ca

Abstract

Crosscutting concerns degrade software quality. Before we can modularize the crosscutting concerns in our programs to increase software quality, we must first be able to find them. Unfortunately, accurately locating the code related to a concern is difficult, and without proper metrics, determining how much the concern is crosscutting is impossible. We propose a systematic methodology for identifying which code is related to which concern, and a suite of metrics for quantifying the amount of crosscutting code. Our concern identification and assignment guidelines resolve some of the ambiguity issues encountered by other researchers. We applied this approach to systematically identify all the requirement concerns in a 13,531 line program. We found that 95% of the concerns were crosscutting—indicating a significant potential for improving modularity—and that our metrics were better able to determine which concerns would benefit the most from reengineering.

1. Introduction and related work

The crosscutting concern problem [13] causes the code related to a concern to be scattered across the program, and often tangled with the code related to other concerns. Several studies indicate that modularizing crosscutting concerns improves software quality [14, 16, 17, 21], providing indirect proof that crosscutting hurts modularity. Unfortunately, there is little guidance for finding crosscutting concerns, and determining when it is profitable to modularize them.

Before we can go about reducing crosscutting code to improve modularity, we must first determine *what* the concerns of the program are (*concern identification*) and *where* they manifest in the program text (*concern assignment*). Only then can we classify the concerns as crosscutting or noncrosscutting.

Alas, manually locating the source code related to a concern is a notoriously hard problem even when the concerns are well defined [2, 15, 18, 22]. Unfortunately, concerns are rarely well defined, partly because the term "concern" is so abstract [20], leading to inconsistent interpretations [6, 15, 18]. Another level of inconsistency is introduced when concerns are assigned to code because existing guidelines [11, 18] are ambiguous. These inconsistencies ensure that experimental results are not repeatable and lead to misguided assessments of the nature and extent of crosscutting in the program.

Automated techniques apply their rules consistently, but they might not find the concerns that the developer is interested in. Execution trace-based techniques, for instance, miss concerns that cannot be isolated by a given test run [6, 7, 22]. These techniques miss "nonfunctional" concerns, such as logging and error handling. Aspect mining [4] and static analysis [19] techniques are useful at generating suggestions for possible concerns, but human interpretation is still required.

We present a novel manual concern identification and assignment methodology in Section 4 that we argue is easier to interpret objectively, resulting in fewer ambiguities than previous approaches [11]. We focus on a manual approach because we would like the concern assignment to cover the entire source base to enable analysis of the full extent of crosscutting.

We also present a novel set of concern metrics in Section 3 based on a formal model (defined in Section 2) for measuring the degree to which the code related to a concern crosscuts the program (*concern quantification*) and the degree to which concerns are separated within a component. We argue that our metrics provide a quantification not possible with existing concern metrics and traditional OO metrics [5]. We include a comparison of some of these metrics in our case study in Section 5. Section 6 concludes.

2. Concern model

Abstractly, a *program specification*, or simply *specification*, is a description of a program. A specification may be *physical*, e.g., a set of *program components*, or *logical*, e.g., a requirements specification or design model. **We define a *concern***

as an element from a program's logical specification. Thus, a logical specification represents a *concern domain* of the program.

The program components that are meaningful depend upon the language in which the program is expressed. Common components for OO programs are files, classes, fields, methods, statements, and statement blocks (for-loops, if-then-else blocks, etc.). A *container component* may contain other components, e.g., a class can contain fields and methods, an if-statement contains one or more statements. In contrast, a *primitive component* does not contain any components (e.g., non-block statements, declarations). A component may *reference* another component, e.g., a method call statement references the method called and a statement that updates a field references that field.

We define our *concern-component mapping* as a tuple $M = (S, T, C_s, C_t, R)$. S and T are the *source specification* and *target specification*, respectively, whose elements can be concerns or components. Concern and component domains may be hierarchical [20], and this hierarchy is described by the C_s and C_t *containment relations*, e.g., $C_s: (s_1, s_2)$, $s_1, s_2 \in S, s_1 \neq s_2$, s_1 is a source element that contains s_2. Finally, R is a *dependency relation* between the two specifications, $R: (s, t), s \in S, t \in T$. This is depicted in Figure 1.

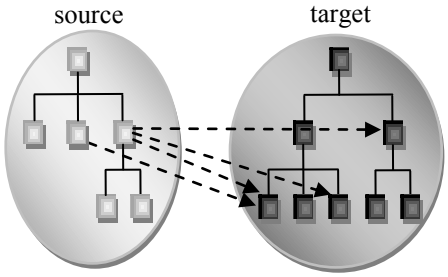

Figure 1. Relation between source and target specifications.

The concern model in [1] is similar, except ours is formally stated using set theory, which facilitates metric definition. Our model also supports hierarchical specifications, making it more general and allowing concern identification, assignment, and quantification to be performed at multiple granularity levels.

For our metrics, we focus on the scenario where S is a set of concerns organized into a hierarchy described by C_s, and T is a set of components (container and primitive) organized according to C_t. Thus our challenge is to determine S and C_s (i.e., concern identification), and R (i.e., concern assignment).

We can now define some common terminology. **A concern is *scattered* if it is related to multiple target elements, and *tangled* if both it and at least one other concern are related to the same target element.** These definitions agree with those in [8]. For the purposes of this paper, **a *crosscutting concern* is a concern that is scattered.** [10, p. 4]

3. Concern metrics

We redefine the *closeness metrics* [22] in terms of our model, and extend them to form the basis for our concern metrics. Our metrics are independent of the particular methodology used to identify and assign concerns.

3.1. Degree of scattering (DOS)

Concentration (*CONC*) measures how many of the source lines related to a concern s are contained within a specific component t (e.g., a file, class, method) [22]:

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

Source lines of code (SLOCs) excludes comments, blank lines, and annotations used for concern assignment. The drawback of CONC is that it does not give a sense for how scattered a concern is and 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 components and $|T| > 1$. DOS is a measure of the variance of the concentration of a concern over all components with respect to the worst case (i.e., when the concern is equally scattered across all components). It has the following properties:

- DOS is normalized to be between 0 (*completely localized*) and 1 (*completely delocalized*, *uniformly distributed*) (inclusive) so that concerns can be meaningfully compared.
- DOS is somewhat *proportional* to the number of components related to the concern.
- DOS is somewhat *inversely proportional* to the concentration. That is, the less concentrated the concern is, the more scattered it is.

A defining characteristic of a module is that its implementation is localized, so a concern that is scattered is by definition not modular. Furthermore, the components across which the implementation of the concern is scattered are less modular than if the scattered concern were not present. We conjecture that the modularity of the program is inversely proportional

Table 1. Concern identification and assignment guidelines.

Concern identification guidelines

CIG1. The concern domain should have objective and definitive membership criteria.
CIG2. The concern domain should be finite.

Concern assignment guidelines based on determining a component-code removal dependency

CAG1. **Primitive components.** Assign a concern to a *primitive component* if and only if the *complete removal of the concern* requires with certainty the removal or modification of the component and its references. *Complete removal* of a concern means no component remains assigned to it (i.e., *disabling* the concern is not enough).
CAG2. **Container components.** If *all* references to and components contained by a *container component* have the same assignment, the container component automatically gets that assignment; otherwise, it is "not assigned" (i.e., the outer source code that encloses the contained components is not uniformly related to any concern.) Alternatively, an assignment to a container component automatically propagates to its contained components. (This is similar to the rules in [18].)
CAG3. **Declarations.** If *all* references to a *declaration* have the same assignment, the declaration automatically gets that assignment; otherwise, it is "not assigned." Alternatively, an assignment to a declaration automatically propagates to its references. For example, *calls* to an assigned method, *uses* of an assigned variable, *references* to an assigned class, etc. are automatically similarly assigned. (This agrees with [11] and [18].)
CAG4. **Subclasses.** If all subclasses of a base class have the same assignment, the base class automatically gets that assignment; otherwise, it is "not assigned." Alternatively, an assignment to a base class propagates to its subclasses.
CAG5. **Virtual methods.** If all overrides of a virtual method have the same assignment, the virtual method automatically gets that assignment; otherwise, it is "not assigned." Alternatively, a virtual method assignment propagates to its overrides.

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) measures how many of the source lines contained within a component *t* are related to concern *s* [22]:

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

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 and $|S| > 1$. DOF is a measure of the variance of the dedication of a component to every concern with respect to the worst case (i.e., when the component is equally dedicated to all concerns). It has the following properties:

- DOF is normalized between 0 (*completely unfocused*) and 1 (*completely focused*) (inclusive) so that components can be meaningfully compared.
- DOF is somewhat *inversely proportional* to the number of concerns related to the component.
- DOF is somewhat *proportional* to the dedication. That is, the more uniformly divided the component's code is among its concerns, the lower its focus.

By averaging the degree of focus (ADOF), we obtain an indication for how well concerns are separated in the program. Ideally, a program should have a low average degree of scattering (ADOS) and a high average degree of focus (ADOF).

Metrics that measure crosscutting are dependent on the concern and component granularity level. Thus, DOS values are only comparable at the same granularity level (similarly for DOF).

4. Concern identification and assignment

Before we can apply DOS to determine how scattered a concern is, we must first identify all the code related to that concern. Similarly, before we can apply DOF to determine how focused a component is, we must first identify the concerns related to all the code in that component.

4.1. Concern identification guidelines

Table 1 presents our guidelines for identifying and assigning concerns. As we mentioned earlier, concerns come from a program's logical specification. For the first guideline in Table 1, *objective* means that two people will identify the same set of concerns. *Definitive* means the question, "Is X *currently* a concern of the program?" has a yes or no answer. This ensures that the domain is well defined and reduces the chance of inconsistency.

The *minimal subset, minimal increment* guidelines proposed by Carver and Griswold [3], while more systematic than most, do not satisfy this criteria because they are not sufficiently objective and

definitive. An independent study found that adherence to their guidelines resulted in inconsistencies [18].

The second guideline further limits the concerns under consideration to be finite. For example, a domain defined as "all future concerns of the program" is not allowed.

4.2. Concern assignment guidelines

Our assignment guidelines in Table 1 are derived from the goals of *software pruning*, *viz.* when removing a concern we would like to remove as many components related to the concern as possible from the program to reduce the program's resource requirements and/or source base. Thus, assignment consists of establishing that a component has a *removal dependency* on the concern.

Previous assignment guidelines (for example, see [9] and [18]) attempt to establish a *contribution relationship*, i.e., a component contributes to the implementation of a concern. In our experience, contribution is hard to decide even when the assignor knows the program well, because it forces the assignor to consider *any possible change to a component that could potentially affect the concern directly or indirectly*. Consider some worst-case examples of a "contribution":

- Removing the `Main` function causes all the concerns of the program to not function properly. (This required its own special case in the identification guidelines in [18].);
- Speeding up some arbitrary piece of code improves the performance of every concern;
- A change to `System.String` could potentially affect every client and derived class.

While these are valid relationships, they are hard to determine, potentially unbounded, and (we argue) not that useful for understanding crosscutting.

In contrast, our guidelines are easier to follow because a) concerns are well defined, b) the range of potential changes that we must consider is limited to "removing a concern," and c) the range of potential affects is limited to the impact on the component under scrutiny. Of course, the assignor must understand the concern and the behavior of the component well enough to judge if a removal dependency exists.

The assignor is free to choose any level of assignment granularity, although this will affect the measurement precision. The guidelines ensure consistency whether the assignment is performed at statement, method, or class level, or higher.

5. Empirical study

To provide initial evidence as to the utility of our metrics, we designed a case study to investigate the following hypotheses:

H1. Our concern metrics are more descriptive than previous concern metrics.
H2. Our concern metrics are more descriptive than traditional OO metrics.

5.1. Case study setup

We chose to evaluate our hypotheses on a medium-sized (13,534 SLOCs, 62 classes) C# program called Goblin, which is a platform for developing virtual reality applications.[1] The rationale for choosing Goblin was that one of the authors was one of the three developers, making concern assignment easier and (we expect) more accurate. To satisfy our identification guidelines we chose the concerns to be the numbered requirements (functional and nonfunctional) taken from Goblin's *software requirement specification* [12].

Of the 137 original requirements, we ignored those that were obviously unrelated to the Goblin platform (e.g., project web site, applications built using Goblin), or which, according to one of the developers, were never implemented. We also removed duplicates and added a few *implicit* requirements:

- Exception/error detection – "Checking the state of a program against a certain predicate when its control flow graph reaches a certain node, at runtime." [9]
- Exception/error handling – The handling of a previously detected (by exception/error detection) erroneous state.
- Clean shutdown – Frees up resources and does not hang/crash on exit.

The final concern domain consisted of 39 requirements.

To test our first hypothesis, we compared *degree of scattering* (DOS) with the *concern diffusion over components* (CDC) metric created by Garcia et al. [9] (see Table 2). The metrics are comparable because they both measure properties of concerns at the class level.

To test our second hypothesis, we compared *degree of focus* (DOF) with the popular *CK metric*, *coupling between object classes* (CBO) [5], which counts the number of classes referenced by a class at compile time. Both metrics measure class dependencies—on classes for CBO, and on concerns for DOF—and are thus comparable.

[1] The case study data (source code, concern mapping, and requirements specification) is available at http://www.cs.columbia.edu/~eaddy/goblin.

Table 2. Garcia and colleague's concern diffusion metrics [9].

Concern Diffusion over Components (CDC)	Counts the number of components that contribute to the implementation of a concern and other components which access them.
Concern Diffusion over Operations (CDO)	Counts the number of methods and advice which contribute to a concern's implementation plus the number of other methods and advice accessing them.
Concern Diffusion over LOC (CDLOC)	Counts the number of transition points for each concern through the LOC. Transition points are points in the code where there is a "concern switch."

5.2. Results

We found that **95% of the requirements were scattered across multiple classes** and 100% across multiple methods. This is consistent with [22], which showed that every feature had no more than 8% of its code in any file. The "Help display" and "Application plug-ins" were the only requirements *not* crosscutting at the class level (according to our definition) since they were completely localized in one class (CDC is 1 and DOS is 0). Table 3 shows the concern metrics for an interesting subset of the requirements concerns.

5.2.1. Comparing concern-based metrics. Looking at the concern diffusion metrics for "Exception/error handling" we notice that exception handling is scattered across 30 classes (CDC) and 107 methods (CDO). Our metrics corroborate this by indicating a relatively high degree of scattering (DOS) across classes (0.80) and methods (0.97). This data reflects the results of other studies [9, 16] that observed that **exception handling is highly scattered**. We also observed that exception handling incurred a high number of "concern switches" (CDLOC is 281, 88th percentile), which has not been reported elsewhere.

However, consider "Monocle-display support" and "Collision detection." Despite the fact that they are both scattered across the same number of classes (CDC is 10), these concerns are actually not scattered equally. The first clue is that Monocle-display has 5 times more source lines. Analyzing the dedication (DEDI) values (not shown) reveal that the bulk (64%) of the source code related to the Monocle-display support requirement is contained in one class, while the bulk of the Collision detection code is evenly split between two classes (33% and 34%). **Whereas CDC fails to make this distinction, our DOS metric indicates that Monocle-display support is less scattered** (DOS of 0.57) than Collision detection (DOS of 0.76) at the class level. This evidence supports our first hypothesis.

Table 3. Concern-based metrics. The metrics include Garcia et al.'s metrics, CDC and CDO, and our degree of scattering (DOS) metrics. Lower values are better (less scattering).

		Diffusion Metrics		*Degree of Scattering*	
Concern (Requirement)	SLOC	CDC	CDO	Class Level	Mthd Level
Graphics API integration	3814	54	501	0.92	0.99
Monocle-display support	2192	**10**	272	**0.57**	0.97
Exception/error detection	513	33	133	0.89	0.98
Exception/error handling	**455**	**30**	**107**	**0.80**	**0.97**
Collision detection	424	**10**	43	**0.76**	0.95
Logging	271	9	26	0.57	0.66
Persistence	190	11	21	0.83	0.92
Clean shutdown	118	13	29	0.83	0.86
Help display	34	1	7	0.00	0.80
Application plug-ins	31	1	6	0.00	0.80

Our concern metrics are more descriptive because they measure the *extent* of scattering, whereas the concern diffusion metrics, and other concern metrics (e.g., [23], [15], and [18]), only measure the *presence* of scattering. Thus, common refactorings, such as consolidating redundant code into a shared function, would not be deemed beneficial by previous metrics. This argument further supports our first hypothesis.

This increased level of precision requires a fine-grained (i.e., statement-level) concern assignment. The difficulty of obtaining such an assignment, especially for large programs, may explain the lack of fine-grained metrics. However, we believe this level of detail is necessary to properly assess the nature and extent of crosscutting for concerns such as exception/error detection and handling and performance.

5.2.2. Comparing concern metrics with traditional OO metrics. Table 4 shows detailed class metrics for a few classes. The CK metrics help us determine, for example, that the Engine class is highly coupled (CBO is 71, 99th percentile). Because it depends on so many classes, we may conclude that the Engine class is complex and fault prone. In contrast, the very low DOF metric (0.06) suggests a *reason* for the complexity: *the Engine class is distracted by too many concerns*. Analyzing both metrics helps us isolate the components with the greatest need for refactoring.

Our concern metrics (DOS and DOF) are more relevant than traditional OO metrics (like the CK metrics) because they relate logical entities from the problem domain (concerns) with physical entities (components). In contrast, the CK metrics only quantify relationships between physical entities (components). Because a change to a program often

Table 4. Class-based metrics. The metrics include the CK metric, CBO, and our degree of focus metric, DOF. Lower values are better for CBO but worse for DOF.

Class	SLOC	CBO	DOF
Framework	1878	106	0.56
Engine	**432**	**71**	**0.06**
ArcBall	**104**	**20**	**1.00**

originates as a change to some concern, our concern metrics allow the impact of that change to be more directly quantified.

6. Conclusion and future work

Before we can modularize concerns, we must be able to locate and quantify them. We presented a systematic methodology for manually identifying concerns and their associated code fragments, which we believe is more accurate and easier to apply consistently than previous approaches. We introduced a suite of metrics for quantifying the degree to which a concern is scattered across components and separated within a component. We showed how our metrics are more descriptive than previous concern metrics and traditional object-oriented metrics.

We plan to further explore the usefulness of our metrics by determining if they actually help predict change impact and other quality indicators. We would like to know if a concern-code mapping helps developers make changes in all the right places. We would like to validate that our concern identification and assignment methodology is consistent, repeatable, and accurate. Ultimately, to realize the full benefits of our metrics and mapping, we need a more practical solution for concern identification and assignment.

7. 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] S. Chidamber and C. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering,* 476-493, 1994.

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

[7] A. D. Eisenberg and K. De Volder, "Dynamic Feature Traces: Finding Features in Unfamiliar Code," *Intl. Conf. on Software Maintenance*, 2005.

[8] 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.

[9] F. C. Filho, N. Cacho, E. Figueiredo, R. Maranhao, A. Garcia, and C. M. F. Rubira, "Exceptions and Aspects: The Devil is in the Details," *Intl. Conf. on Foundations of Software Engineering*, 2006.

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

[11] A. Garcia, C. Sant'Anna, C. Chavez, S. Viviane, C. Lucena, and A. v. Staa, "Agents and Objects: An Empirical Study on Software Engineering," Technical Report 06-03, CS Dept, PUC-Rio, 2003.

[12] IEEE, "IEEE recommended practice for software requirements specifications," *IEEE Std 830-1998,* 1998.

[13] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar, "Aspect-oriented programming," *ACM Computing Surveys,* 28(4es):154, 1996.

[14] U. Kulesza, C. Sant'Anna, A. Garcia, R. Coelho, A. V. Staa, and C. Lucena, "Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study," *Intl. Conf. on Software Maintenance*, 2006.

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

[16] M. Lippert and C. V. Lopes, "A study on exception detecton and handling using aspect-oriented programming," *Intl. Conf. on Software Engineering*, 2000.

[17] C. V. Lopes and S. Bajracharya, "An Analysis of Modularity in Aspect-Oriented Design," *Aspect-Oriented Software Development*, 2005.

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

[19] M. P. Robillard and G. C. Murphy, "Automatically Inferring Concern Code from Program Investigation Activities," *Automated Software Engineering*, 2003.

[20] S. M. Sutton, Jr. and I. Rouvellou, "Concern Modeling for Aspect-Oriented Software Development," in *Aspect-Oriented Software Development*, R. E. Filman, T. Elrad, S. Clarke, and M. Aksit, eds. Boston, MA: Addison-Wesley, 2005, pp. 479-505.

[21] 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.

[22] 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.

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