

On the Relationship between Crosscutting Concerns and Defects: An Empirical Investigation

Marc Eaddy, Vibhav Garg,
Alfred Aho
Columbia University
New York, New York
{eaddy, vgarg, aho}@
cs.columbia.edu

Nachiappan Nagappan
Microsoft Research
Redmond, Washington
nachin@microsoft.com

Kaitlin Duck Sherwood
University of British Columbia
Vancouver, British Columbia
ducky@webfoot.com

Abstract

Empirical studies indicate that crosscutting concerns negatively impact internal quality indicators (e.g., cause increased coupling). However, empirical evidence indicating that crosscutting negatively impacts external quality indicators is lacking. To address this, we present the results of an experiment to determine if concerns whose implementations are scattered are more likely to have defects. Our results provide preliminary evidence that scattering is strongly correlated with defects at statistically significant levels.

1. Introduction

It is hard to write reliable software. Defects creep in at every stage of the development process, avoid detection during testing, and eventually escape to the customer. Enormous effort goes into safe development techniques, program analysis, and prerelease testing to reduce the number of defects in a delivered software system. To better direct these efforts, we need a way to estimate where defects are likely to occur.

We consider the possibility that crosscutting concerns are a likely source of defects. A *crosscutting concern* is a concern of the program (e.g., feature, requirement) whose implementation is scattered across the program and often tangled with the source code related to other concerns.¹ Several empirical studies [19] [16] [28] [36] [38] argue that crosscutting concerns degrade code quality because they negatively impact *internal quality metrics* (i.e., measures derived from the program itself [25]), such as program size, coupling, and separation of concerns.

¹ For this study, we consider a crosscutting concern to be synonymous with a scattered concern, agreeing with the definition in [17].

But do these negative impacts on internal quality metrics also result in negative impacts on external quality? Internal metrics are of little value unless there is convincing evidence that they are related to important externally visible quality attributes [22], such as maintenance effort, field reliability, and observed defects [13].

We argue in this paper that crosscutting concerns might negatively impact at least one external quality attribute—defects. Our theoretical basis is that a crosscutting concern is harder to implement and change consistently because multiple locations in the code (that may not be explicitly related) have to be updated simultaneously. Furthermore, crosscutting concerns may be harder to understand because developers must reason about code that is distributed across the program, and must mentally untangle the code from the code related to other concerns.

Some controlled experiments on program comprehension suggest our theory is valid. Letovsky and Soloway use the term *delocalized plan* to refer to a concern whose implementation is “realized by lines scattered in different parts of the program.” They observed that programmers had difficulty understanding delocalized plans, and this resulted in several kinds of incorrect modifications [27]. Robillard, Coelho, and Murphy observed that programmers made incorrect modifications when they failed to account for the scattered nature of the concern they were modifying:

“Unsuccessful subjects made all of their code modifications in one place even if they should have been scattered to better align with the existing design.” [33]

Other studies (e.g., [20] [8]) indicate that programmers make mistakes when modifying classes whose implementations are scattered due to inheritance.

Finally, enhancements or bug fixes applied to a crosscutting concern may induce changes in multiple

source files, leading to increased *code churn*. Nagappan and Ball showed that code churn is a good predictor of system defect density [30], and we propose that changes to crosscutting concerns may be the root cause.

A strong relationship between crosscutting and defects, if it exists, indicates the need for further studies to determine the root cause of the relationship: Are changes to crosscutting concerns more likely to be applied inconsistently? Are crosscutting concerns inherently more difficult to understand?

This paper is organized in the classic way. After discussing the state of the art for assessing the impact of crosscutting concerns on code quality (Section 2), we describe the design of our study (Section 3). Our results are reported in Section 4. In Section 5 we discuss threats to validity. Section 6 concludes.

2. Background and Related Work

Several researchers have studied the impact of crosscutting concerns on code quality. Most of the effort has concentrated on developing new internal metrics, or adapting existing ones, for quantifying crosscutting, and assessing the impact of modularizing crosscutting concerns using techniques such as aspect-oriented programming.

2.1. Concern Metrics

Several researchers (e.g., [38], [26], and [32]) have created concern metrics that measure scattering in absolute terms (e.g., number of classes that contribute to the implementation of the concern). For example, Garcia and colleagues used their *concern diffusion metrics* in several studies (e.g., [19] [16]) to show that, in general, modularizing crosscutting concerns using aspect-oriented programming improves separation of concerns.

In Section 3.6, we describe our *degree of scattering metrics*, which we believe complement the concern diffusion metrics by providing a more fine-grained measurement of scattering. We include both sets of metrics in our correlation results for comparison and to determine which metric is most strongly correlated with defects.

We know of one study besides our own that correlates aspect-/concern- related metrics with external quality attributes. Bartsch and Harrison examined change history data for a set of aspects and found a statistically significant correlation between aspect coupling and maintenance effort [2]. Their metrics were different from ours (aspect coupling versus concern scattering), and their external quality

indicator was different (effort versus defects). Whereas we investigated the impact of a crosscutting concern on code quality *prior to* refactoring using aspects, they looked at the impact *after* refactoring. A benefit of our scattering metrics is that they may help identify the crosscutting concerns that would benefit the most from refactoring.

2.2. Correlating Metrics with Defects

Several researchers have attempted to find a relationship between defects and internal product metrics, such as code churn [30], size metrics [14] [9] [21], object-oriented metrics (e.g., the CK metrics [10]) [9] [21], and design metrics [9]. We add to this body of research by examining the relationship between concern metrics and defects.

2.3. Mining Software Repositories

In recent years, researchers have learned to exploit the vast amount of data that is contained in software repositories such as version and bug databases [29] [30] [31] [39]. The key idea is that one can map *problems* (in the bug database) to *fixes* (in the version database) and thus to those locations in the code that caused the problem [11] [18] [35]. In Section 3.4, we describe how we use these techniques to systematically map defects to methods and fields.

3. Study Design

The goal of our study was to answer the questions: Are crosscutting concerns defect prone? Which scattering metric is the best at predicting defects? Which scattering level (e.g., class level, method level) is the most appropriate for predicting defects? Finally, which is more important for predicting defects: the size of the concern implementation (in terms of lines of code), i.e., *concern size*, or how much that implementation is scattered?

3.1. Research Hypotheses

We formalize these questions to form our research hypotheses:

	Hypothesis
H ₁	The more scattered a concern's implementation is, the more likely it is to have defects.
H ₂	Our <i>degree of scattering</i> measures are more strongly correlated with defect count than direct (absolute) scattering measures.

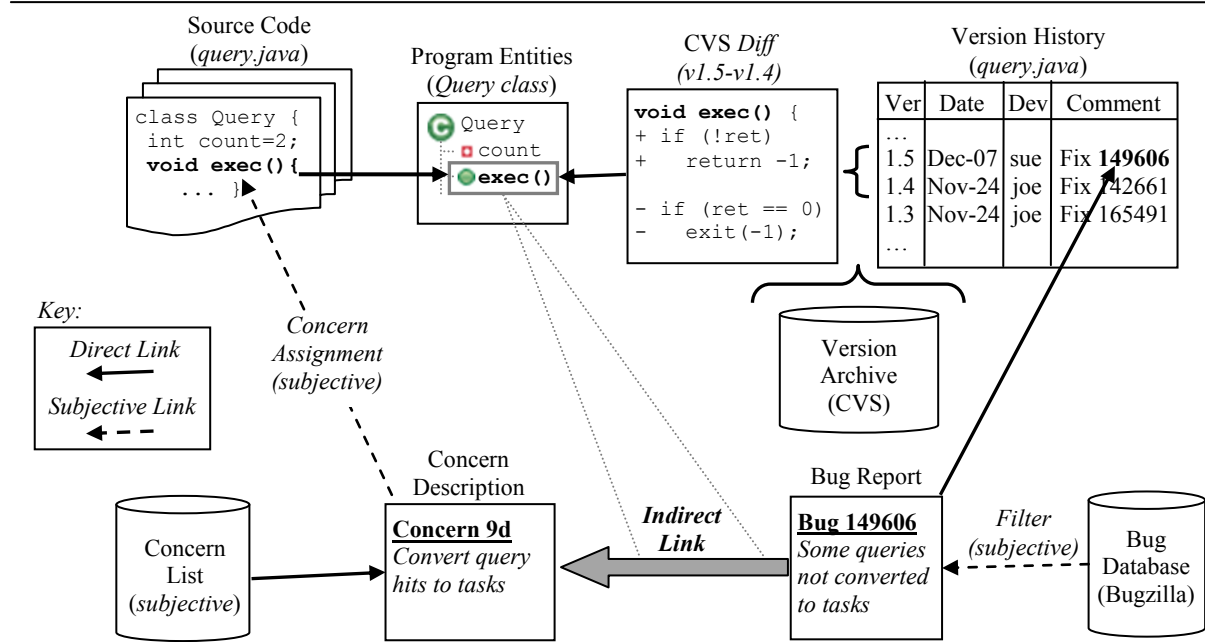


Figure 1. Our methodology for associating defects with concerns in Mylar-Bugzilla

H ₃	Scattering at the method level is more strongly correlated with defect count than at the class level.
H ₄	Scattering is a better predictor of defects than the concern size.

3.2. Our Case Studies

For our experiment we looked at two open source projects: Mylar and dbviz. Mylar² is a commercial-grade plug-in for the Eclipse³ integrated development environment that enables a task-focused development methodology [24]. It was developed by a team of experienced Ph.D. students and professional developers in conjunction with one of the authors of this paper. Version 1.0.1 consists of 168,457 lines of Java code (LOCs); however, we limited our analysis to two components: bugzilla.core and bugzilla.ui, totaling 56 classes, 427 methods, 457 fields, 13,649 lines of Java code. From here on, we refer to this subset as Mylar-Bugzilla.

Dbviz⁴ is a database schema visualization program developed by students for a software engineering class at University of Illinois at Urbana-Champaign. Version 0.5 consists of 77 classes, 458 methods, 391 fields, and 12,744 lines of Java code.

We selected these projects in part because they had a source code version archive (CVS), a bug database (Bugzilla⁵ for Mylar-Bugzilla, SourceForge⁶ for dbviz), were well documented, and were of modest size. This last criterion was important because of our manual concern and bug assignment methodology.

Figure 1 depicts our operational methodology for assigning defects to concerns. The following sections describe the methodology in detail.

3.3. Concern Identification Methodology

Our first step was to identify the concerns for each program. To reduce inconsistency during concern assignment, it is important that the concerns be well defined [12]. We also want the concerns to cover most of the source code (or subset, in the case of Mylar-Bugzilla) so that we can capture most of the defects.

We relied on the project documentation and our familiarity with the projects to help us specify the concerns. The requirements for the Mylar-Bugzilla component were reverse engineered based on the “New and Noteworthy” section of the Mylar web site and personal experience with the development and usage of the component. For dbviz, the requirements were organized as a set of use case documents.

² <http://www.eclipse.org/mylar>

³ <http://www.eclipse.org>

⁴ <http://jdbv.sourceforge.net/dbviz>

⁵ <http://www.bugzilla.org>

⁶ <http://www.sourceforge.net>

We used 28 of Mylar-Bugzilla’s functional and nonfunctional requirements related to the bugzilla.core and bugzilla.ui components (i.e., *requirement concerns*), and dbviz’s 13 use cases (i.e., *use-case concerns*).

Ideally, the project selection criteria would have included the requirement that all projects have the same concern domain (e.g., requirements, use cases). However, we had difficulty finding projects that met all of these criteria.

3.4. Mapping Concerns to Source Code

We used the ConcernMapper [34] plug-in for Eclipse to manually assign methods and fields in the source code to one or more concerns. We extended ConcernMapper to store mapping data in a database and to calculate the concern metrics.

To determine if a method or field was related to a concern, we required (1) that the method be executed or the field read/written during the execution of a hypothetical test case for the concern [37], and (2) that the element would need to be removed or changed if we wanted to completely remove the concern from the program [12]. The first test associates the concern with a large number of elements that are conceptually unrelated (e.g., Main, String.append). The second test is more subjective and attempts to prune the list of nonessential elements (e.g., generic elements or elements shared by many concerns). A side effect of the second test is that it automatically excludes libraries since the methods and fields contained therein cannot be changed or removed (because we do not have the source code).

Our assignment methodology results in some classes, methods, and fields not being mapped because they are not specific to any concern. In addition, limitations of our tool prevented us from assigning inner classes and enumeration types. Table 1 summarizes the project and mapping statistics.

3.5 Mapping Defects to Concerns

Each project had a source code *version archive* (CVS) for keeping track of source code changes and a *bug database* (e.g., *Bugzilla*) for keeping track of defects⁷. Effort was made by the project teams to avoid reporting duplicate defects.

We define the term *defect* as an error in the source code, and the term *failure* to refer to an observable error in the program behavior. In other words, every failure can be traced back to some defect, but a defect

need not result in a failure. The bug databases did not distinguish between *prerelease defects* (e.g., defects discovered during development and testing) and *postrelease defects* (e.g., defects that cause failures observed by a customer).

A *defect fix* is the set of lines in the source code (which may span multiple files or even multiple versions of the same file) added, removed, or modified to fix a defect. Our underlying assumption is that it is reasonable to associate a defect with a concern if the source code that implements the concern must be changed to fix the defect.

Unfortunately, although both projects had a version archive, the amount of detail available in the change logs for understanding the reason for a change varied considerably. Because of this, we used two different techniques for associating defects with concerns.

Mylar-Bugzilla. Every change in Mylar-Bugzilla is automatically associated with an entry in the bug database. This is because Mylar-Bugzilla was developed using Mylar. When a developer begins working on a bug fix, they use the Mylar Task List to first indicate that the bug is active [24]. When they check-in their changes, Mylar automatically creates a commit message based on the bug that was active:

“RESOLVED - bug 149606: Some queries not converted to tasks. https://bugs.eclipse.org/bugs/show_bug.cgi?id=149606.”

We relied on change history data to assign defects to program entities (e.g., methods and fields). We only considered fixed defects (i.e., their *resolution* was “fixed” and their *severity* was not “enhancement”). This limitation was necessary for us to automatically associate defects with bug fixes (i.e., we did not have to figure out how to fix the bug ourselves). This filter netted 1368 defects. We excluded defects that did not require changes to bugzilla.core or bugzilla.ui, which left us with 110 defects.

To determine which program entities were associated with a defect fix, we manually inspected the lines that were added, removed, or modified. We ignored inconsequential changes to whitespace, comments, and element names, and changes to

Table 1. Descriptive statistics for projects and mappings

Entity	Mylar-Bugzilla			dbviz		
	All	Mapped	%	All	Mapped	%
Classes	56	44	79	77	59	77
Methods	427	253	59	458	280	61
Fields	457	230	50	391	204	52
Lines	13649	5914	43	12744	4672	37
Defects	110	101	92	56	47	84

⁷ We use the terms *defect* and *bug* interchangeably.

Table 2. Concern-based metrics

Lines of Code for Concern (LOCC)	Number of lines of code that contribute to the implementation of a concern (i.e., number of lines of <i>concern code</i>).
Concern Diffusion over Components (CDC)	Number of classes that contribute to the implementation of a concern and other classes and aspects which access them [16].
Concern Diffusion over Operations (CDO)	Number of methods which contribute to a concern’s implementation plus the number of other methods and advice accessing them [16].
Degree of Scattering over Classes (DOSC)	Degree to which the concern code is distributed across classes. When DOSC is 0 all the code is in one class. When DOSC is 1 the code is equally divided among all the classes. See the example in Figure 2. [12]
Degree of Scattering over Members (DOSM)	Degree to which the concern code is distributed across methods and fields. Varies from 0 to 1 similar to DOSC. [12]

methods and fields that were later removed (i.e., did not exist in the latest version). (We discuss the implications of this in Section 5.)

Using this process we were able to objectively map all 110 defects to program entities. After both mappings were complete⁸, we assigned bugs to concerns if they shared a method or field in common. In other words, a bug was assigned to a concern if the bug fix was *tangled* (shared a program entity) with the concern. However, we had to leave out 9 defects because they mapped to program entities not associated with any concern. This resulted in 101 defects mapped to 28 requirements concerns.

Dbviz. For dbviz, commit messages were written by the developer. Here is a typical example:

“Fixed a bug where adding a table via the ConnectionList wasn’t undoing properly.”

Although we could have used information retrieval techniques to associate defect descriptions with change log entries [35] and then with concerns, we instead relied on our judgment to associate defects directly with concerns. There were a few cases where defect identifiers were used in the commit message, which allowed us to validate some of our defect assignments; however, they were not used consistently. Despite these issues, we found the assignment to be straightforward.

Since we were not able to automatically associate defects with defect fixes, we included “open” and “closed” defects, which totaled 56 defects. We ignored 1 defect that was a general refactoring request (“Move action classes to dedicated package”) and 8 defects that were associated with features not covered by any use case.⁹ This resulted in 47 defects mapped to 13 use-case concerns.

⁸ To eliminate bias, a different person performed the mapping of concerns to program entities.

⁹ For example, “zooming,” “about box,” and “logging,” were not covered by any use case.

3.6. Concern Metrics

A concern is *scattered* (i.e., crosscutting) if it is implemented by multiple program elements [4] [12] [15]. We investigated a concern-based size metric and four different scattering metrics to determine which (if any) had the strongest correlation with defects. Table 2 provides a description of the metrics.

The *concern diffusion metrics*, created by Garcia and colleagues, measure scattering in absolute terms as a count of the number of classes (CDC) or methods (CDO) that implement the concern [16].¹⁰ Our *degree of scattering metrics* (DOSC and DOSM) provide more information by further considering how the concern’s code is distributed among the elements [12]. Our scattering metrics build upon the *concentration metric* (CONC) introduced by Wong et al. [37].

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

where t is a class when measuring DOSC and a method when measuring DOSM. Since we assigned concerns at the method level, we counted the LOCs for the method as relating to the concern.

Degree of scattering 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):

$$DOS(s) = 1 - \frac{|T| \sum_{t \in T} \left(CONC(s, t) - \frac{1}{|T|} \right)^2}{|T| - 1}$$

For DOSC, T is the set of classes in the program, and for DOSM, the set of methods.

The difference between CDC and DOSC is illustrated in Figure 2. The pie chart shows how the

¹⁰ They can also measure constructs from aspect-oriented programming such as aspects and advice.

code related to the concern is distributed among four classes. In the first scenario, the implementation is evenly divided among the four classes (the worst case). The DOSC value is

$$DOSC = 1 - \frac{|4| \left(\left(.25 - \frac{1}{|4|} \right)^2 + \left(.25 - \frac{1}{|4|} \right)^2 + \left(.25 - \frac{1}{|4|} \right)^2 + \left(.25 - \frac{1}{|4|} \right)^2 \right)}{|4| - 1}$$

$$= 1 - 0 = 1$$

In the second scenario, the DOSC value is

$$DOSC = 1 - \frac{|4| \left(\left(.97 - \frac{1}{|4|} \right)^2 + \left(.01 - \frac{1}{|4|} \right)^2 + \left(.01 - \frac{1}{|4|} \right)^2 + \left(.01 - \frac{1}{|4|} \right)^2 \right)}{|4| - 1}$$

$$= 0.08$$

DOSC is close to 0, indicating the implementation in the second scenario is almost completely localized (0 means completely localized). CDC cannot distinguish the two implementations, as evident by the value of 4 for both. Our hypothesis (H_2) is that degree of scattering more accurately quantifies the modularity of a concern, and so should be a better predictor of defects than absolute scattering metrics such as CDC and CDO.

4. Results and Discussion

In this section we discuss the results for the four hypotheses. Each hypothesis is discussed in a separate subsection.

4.1. Is Scattering Correlated with Defect Count?

Table 3 presents our Spearman correlation results for the Mylar project. For Mylar, all scattering metrics (DOSC, DOSM, CDC, and CDO) have positive correlations with defects, and CDC and CDO are strongly correlated (.569 and .609). All correlations are statistically significant at the 5% level, that is, there is a 95% probability that (for Mylar) a relationship exists. As with all statistical analysis results, we cannot assume this correlation will hold for other projects or even for future versions of Mylar. Only when more studies replicate our results can we become confident that the relationship is real.

For dbviz (see Table 4), the small sample size—only 13 use-case concerns—severely limited the statistical significance of our results. For this project, we were not able to learn anything about the relationship between scattering and defects.

From both tables we observe that the scattering metrics are strongly correlated with each other. This is expected since CDC and CDO are coarser versions of

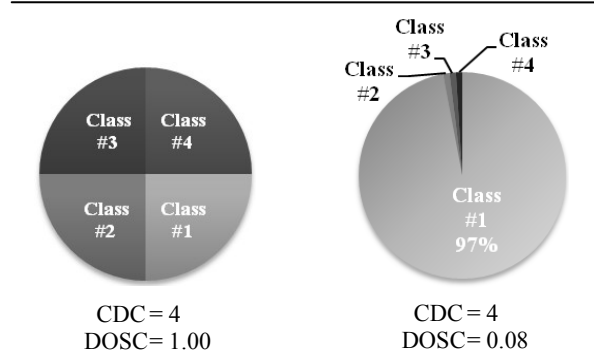


Figure 2. Comparing CDC and DOSC for two different implementations of the same concern

DOSC and DOSM. In addition, the member-level metrics were strongly correlated with their class-level counterparts. This is also expected since a class is only associated with a concern if at least one of its members is associated.

In summary, the Mylar case study showed a positive statistically significant correlation between scattering and defects, which supports our hypothesis H_1 :

Concern scattering is strongly correlated with defects.

4.2. Which Scattering Metric has the Strongest Correlation with Defect Count?

For Mylar, CDC and CDO were more strongly correlated with defects than DOSC and DOSM. Thus, we weakly reject our hypothesis H_2 .

Absolute scattering measures are more strongly correlated with defects than degree of scattering measures.

We were somewhat surprised by this result. We expected DOSC and DOSM to decidedly outperform CDC and CDO because we believe degree of scattering more faithfully quantifies the scattered nature of a concern.

It is possible that the way we chose concerns was a factor. For Mylar, we created an initial list of requirements and updated it as we did the mapping. This can lead to a kind of *concern identification bias* whereby the assignor may choose concerns based on how well they align with the code.¹¹ In contrast, for

¹¹ We have also witnessed this phenomenon in a previous study [12].

Table 3. Correlation matrix for Mylar. Shows how the concern metrics correlate with each other and with defects. Bold values indicate a statistically significant correlation. The sample size N (number of concerns) is 28.

		DOSC	DOSM	CDC	CDO	LOCC	Defects
DOSC	Correlation Coefficient Sig. (2-tailed)	1.000	.641 ($<.0005$)	.844 ($<.0005$)	.568 (.002)	.375 (.049)	.389 (.041)
DOSM	Correlation Coefficient Sig. (2-tailed)		1.000	.771 ($<.0005$)	.909 ($<.0005$)	.626 ($<.0005$)	.498 (.007)
CDC	Correlation Coefficient Sig. (2-tailed)			1.000	.779 ($<.0005$)	.653 ($<.0005$)	.569 (.002)
CDO	Correlation Coefficient Sig. (2-tailed)				1.000	.706 ($<.0005$)	.609 (.001)
LOCC	Correlation Coefficient Sig. (2-tailed)					1.000	.767 ($<.0005$)
Defects	Correlation Coefficient Sig. (2-tailed)						1.000

dbviz we used existing use cases. The results bear out our conjecture: the average DOSC (ADOSC) was much higher in dbviz (.787) than Mylar (.362). Clearly, the way concerns are defined significantly impacts scattering measures.

Another factor is that we only analyzed a portion of Mylar, i.e., Mylar-Bugzilla. It is likely that the scattering measures for the concerns would change if we took into account the entire Mylar codebase. Moreover, only 110 of the 1368 (8%) fixed defects in Mylar required changes to Mylar-Bugzilla. This begs the question: which concerns are associated with the remaining 1258 defects? It is likely that the defect counts for the concerns would change significantly if we took into account all of Mylar.

4.3. What Level of Scattering has the Strongest Correlation with Defect Count?

Although the difference is slight, for Mylar, method-level scattering was more strongly correlated with defects than class-level scattering when comparing DOSM (.498) versus DOSC (.389) and CDO (.609) versus CDC (.569). Nevertheless, we weakly accept our hypothesis H_3 :

Scattering at the method level appears to be more strongly correlated with defects than at the class level.

4.4. Is Concern Size A Better Predictor of Defect Count than Scattering?

For Mylar, the size of the concern implementation (LOCC) alone was the best predictor of defect count (the correlation is .767 at a 1% significance level). This is consistent with several other studies ([9], [21], and [6]) that found a statistically significant relationship between size metrics and defects. This indicates that larger concerns have more defects. However, this is not a foregone conclusion, since a study conducted by Fenton and Ohlsson [14] found that size metrics were not good predictors of fault-prone modules.

Size is an important metric as it is often correlated with other internal product metrics [13]. For example, if a concern has a large number of classes involved in its implementation (CDC), this usually implies a large number of lines (LOCC). In fact, CDC and CDO cannot increase without a simultaneous increase in LOCC. Looking at Tables 3 and 4, we indeed see a high correlation: for Mylar-Bugzilla, CDC-LOCC is .653 and CDO-LOCC is .706, and for dbviz, CDO-LOCC is .885.

The DOSC and DOSM equations are not directly dependent on the number of lines associated with a concern, but rather on how those lines are distributed across classes and methods. Despite this, for Mylar-Bugzilla DOSC and DOSM are correlated with LOCC, although the correlation is not as strong as CDC and CDO (.389 for DOSC and .498 for DOSM), and for

Table 4. Correlation matrix for dbviz. Shows how the concern metrics correlate with each other and with defects. Bold values indicate a statistically significant correlation. The sample size N (number of concerns) is 13.

		DOSC	DOSM	CDC	CDO	LOCC	Defects
DOSC	Correlation Coefficient Sig. (2-tailed)	1.000	.588 (.035)	.863 ($<.0005$)	.537 (.058)	.289 (.338)	.188 (.539)
DOSM	Correlation Coefficient Sig. (2-tailed)		1.000	.694 (.008)	.920 ($<.0005$)	.878 ($<.0005$)	.419 (.154)
CDC	Correlation Coefficient Sig. (2-tailed)			1.000	.635 (.020)	.507 (.077)	.510 (.075)
CDO	Correlation Coefficient Sig. (2-tailed)				1.000	.885 ($<.0005$)	.536 (.059)
LOCC	Correlation Coefficient Sig. (2-tailed)					1.000	.528 (.064)
Defects	Correlation Coefficient Sig. (2-tailed)						1.000

dbviz, DOSM is highly correlated with LOCC (.878). This could mean that for Mylar-Bugzilla, it just so happens that larger concerns are distributed more uniformly.

If concern size has the highest correlation with defects, why consider scattering at all? In fact, the high correlations between scattering and size might indicate that scattering is a surrogate for size, i.e., scattering and LOCC may be collinear. To evaluate this possibility, we created four separate predictive models:

1. Linear regression using only LOCC,
2. Multiple linear regression using only scattering metrics (DOSC, DOSM, CDC, and CDO),
3. Multiple linear regression using ALL concern metrics, and
4. Multiple linear regression using components obtained via Principal Component Analysis [23].

R^2 [5] is a measure for how well the model accounts for the defect count given the dataset (e.g., the number of defects associated with the concerns in Mylar-

Bugzilla). It cannot be interpreted as the quality of the dataset to make future predictions. Adjusted R^2 accounts for any bias in the R^2 measure by taking into account the degrees of freedom of the predictor variables (e.g., DOSC) and the sample population.

The results in Table 5 indicate that a model built using LOCC alone is a better predictor (Adj. R^2 is .720) of defect count than a model built using all the scattering metrics (Adj. R^2 is .675), although the difference in predictive power is not very significant. The most accurate model combined all metrics (Adj. R^2 is .753). Based on this data, we weakly reject our hypothesis H_4 —more studies are needed before we can make stronger conclusions:

Concern size is a slightly better predictor of defects than scattering; however, the best predictor combines size and scattering.

Although concern size is a better predictor of defect count than scattering, scattering may be important to consider for other reasons, e.g., for identifying crosscutting concerns that weaken the modularity of the program.

5. Threats to Validity

There are several factors that affect the validity of our experiment.

Table 5. Regression fit for four predictive models

Model	R^2	Adjusted R^2	Std. Error
LOCC only	.730	.720	3.88488
Scattering metrics	.723	.675	4.18562
All metrics	.799	.753	3.64583
Principal components	.775	.747	3.68906

5.1. Internal Validity

In the Mylar study, 9 of the 110 defects were mapped to methods or fields not covered by any concern. In most cases, this is not an issue since a program element may be related to a concern from a different concern domain (e.g., “resource deacquisition” is a programming concern rather than a requirement or design concern). However, it may also mean that some concerns were not accounted for, which can skew the measurements.

Many of the Mylar defects were clearly enhancements although they were not classified as such. This can lead to inflated concern defect counts. Although the change history indicated which defect was fixed, in some cases unrelated modifications were included with the defect fix [7], which can lead to false positives where a defect is associated with the wrong concern.

There were many code changes that involved methods and fields that were not present in the latest version because they were removed or renamed. This can result in a false negative where a defect should have been associated with a concern but was not. To reduce these we would need a concern mapping for every revision, not just the latest.

Finally, our measurement tool only counts the lines of code associated with methods and fields. It does not exclude comments or whitespace inside the methods, nor does it count lines associated with inner class, enumeration type, and namespace declarations, and import statements. This explains the low code coverage for Mylar (43%) and dbviz (37%) shown in Table 1. This can result in higher LOCC counts that can skew results, or lower counts that can produce false negatives (i.e., a bug should have been mapped to a concern but was not).

5.2. External Validity

Studies (e.g., [26] [32]) have shown that concern identification and assignment is highly subjective, which limits the repeatability of our results and their applicability to other projects. Our future work is to incorporate a more automated concern assignment methodology (e.g., [1]).

As stated by Basili et al., drawing general conclusions from empirical studies in software engineering is difficult because any process depends to a large degree on a potentially large number of relevant context variables. For this reason we cannot assume that the results of a study generalize beyond the specific environment in which it was conducted [3].

6. Conclusions and Future Work

Ours is the first study that associates concerns with defects and uses statistical analysis to correlate concern metrics with defects. We examined the concerns of two medium-size projects and found that the more scattered the implementation of a concern is, the more likely it is to have defects. All the measures we investigated were effective predictors of defects. We also found that the size of the concern alone, in terms of lines of code, was the strongest predictor of defects.

This evidence, although preliminary, is important for several reasons. It adds credibility to the claims about the dangers of crosscutting made by the aspect-oriented programming and programming language communities. By establishing a relationship between concern metrics and an external quality indicator, we provide a stronger form of validation for these metrics than previous empirical studies (e.g., [19] [16]).

It is important to realize that the novelty of our experiment and the subjectivity inherent in our methodology, severely limit the applicability of our results. Further studies are needed before we can attempt to draw generalizable conclusions about the relationship between scattering and defects.

Several questions remain. Can we reduce the likelihood of defects by reducing crosscutting, or is the likelihood constant regardless of how the concern is implemented? What is the relationship between code churn and scattering? If a relationship exists, we can use code churn to help identify crosscutting concerns [7] and as a cost-effective surrogate for measuring scattering. When code churn levels are dangerously high, concern analysis may provide an explanation and an actionable plan for reducing churn (i.e., modularize the underlying crosscutting concerns).

All the data for the case studies is available at http://www.cs.columbia.edu/~eaddy/mylar_study.

ACKNOWLEDGMENTS

This paper would not have been possible without the help of Gail C. Murphy, who arranged the Mylar-Bugzilla study and provided excellent feedback.

References

- [1] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, E. Merlo, "Recovering Traceability Links between Code and Documentation," *IEEE Trans. Soft. Eng.*, 28(10):970-983, October 2002.
- [2] M. Bartsch, R. Harrison, "Towards an Empirical Validation of Aspect-Oriented Coupling Measures," in *Workshop on Assessment of Aspect Techniques (ASAT 2007)*, March 12 2007.

- [3] V. Basili, F. Shull, F. Lanubile, "Building Knowledge through Families of Experiments," *IEEE Transactions on Software Engineering*, 25(4):456-473, 1999.
- [4] K. v. d. Berg, J. M. Conejero, J. Hernández, "Analysis of Crosscutting across Software Development Phases based on Traceability," in *Wkshp. on Aspect-Oriented Requirements Eng. and Arch. Design (Early Aspects 2006)*, May 21 2006.
- [5] N. Brace, R. Kemp, R. Snelgar, *SPSS for Psychologists*. Hampshire, UK: Palgrave Macmillan, 2003.
- [6] L. Briand, J. Wuest, J. Daly, V. Porter, "Exploring the Relationships Between Design Measures and Software Quality in Object Oriented Systems," *Journal of Systems and Software*, 51:245-273, 2000.
- [7] G. Canfora, L. Cerulo, M. D. Penta, "On the Use of Line Co-change for Identifying Crosscutting Concern Code," in *International Conference on Software Maintenance (ICSM 2006)*, Sept 24-27 2006.
- [8] M. Cartwright, "An empirical view of inheritance," *Information and Software Technology*, 40:795-799, 1998.
- [9] M. Cartwright, M. Shepperd, "An Empirical Investigation of an Object-Oriented Software System," *IEEE Transactions on Software Engineering*, 26(8):786-796, 2000.
- [10] S. Chidamber, C. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*:476-493, 1994.
- [11] D. Čubranić, G. C. Murphy, "Hipikat: Recommending Pertinent Software Development Artifacts," in *Intl. Conf. on Software Engineering (ICSE 2003)*, 2003, pp. 408-418.
- [12] M. Eaddy, A. Aho, G. C. Murphy, "Identifying, Assigning, and Quantifying Crosscutting Concerns," in *Workshop on Assessment of Contemporary Modularization Techniques (ACoM 2007)*, May 22 2007.
- [13] K. E. Emam, "A Methodology for Validating Software Product Metrics," Tech. Rep. NCR/ERC-1076, National Research Council of Canada, June 2000.
- [14] N. E. Fenton, N. Ohlsson, "Quantitative analysis of faults and failures in complex software systems," *IEEE Transactions on Software Engineering*, 26(8):797-814, 2000.
- [15] E. Figueiredo, A. Garcia, C. Sant'Anna, U. Kulesza, C. Lucena, "Assessing Aspect-Oriented Artifacts: Towards a Tool-Supported Quantitative Method," in *Wkshp. on Quant. Approaches in OO Soft. Eng. (QAOOSE 2005)*, July 2005.
- [16] F. C. Filho, N. Cacho, E. Figueiredo, R. Maranhao, A. Garcia, C. M. F. Rubira, "Exceptions and Aspects: The Devil is in the Details," in *Intl. Conf. on Foundations of Software Engineering (FSE 2006)*, 2006, pp. 152-162.
- [17] R. E. Filman, T. Elrad, S. Clarke, M. Aksit, *Aspect-Oriented Software Development*. Boston, MA: Addison-Wesley, 2005.
- [18] M. Fischer, M. Pinzger, H. Gall, "Populating a Release History Database from Version Control and Bug Tracking Systems," in *International Conference on Software Maintenance (ICSM 2003)*, 2003, pp. 23-32.
- [19] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, A. v. Staa, "Modularizing Design Patterns with Aspects: A Quantitative Study," in *Aspect Oriented Software Development (AOSD 2005)*, March 14-18 2005.
- [20] R. Harrison, S. Counsel, R. Nithi, "Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems," *Journal of Systems and Software*, 52:173-179, 2000.
- [21] R. Harrison, L. Samaraweera, M. Dobie, P. Lewis, "An Evaluation of Code Metrics for Object-Oriented Programs," *Information and Software Technology*, 38:443-450, 1996.
- [22] ISO/IEC, "IDS 14598-1 Information Technology - Software Product Evaluation," 1996.
- [23] E. J. Jackson, *A User's Guide to Principal Components*. New York, NY: John Wiley & Sons, Inc., 1991.
- [24] M. Kersten, G. C. Murphy, "Using task context to improve programmer productivity," in *Foundations of Software Engineering (FSE 2006)*, November 2006.
- [25] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, J. P. Hudepohl, "Classification-Tree Models of Software Quality Over Multiple Releases," *IEEE Transactions on Reliability*, 49(1):4-11, 2000.
- [26] A. Lai, G. C. Murphy, "The Structure of Features in Java Code: An Exploratory Investigation," in *Wkshp. on Multi-Dimensional Sep. of Concerns (OOPSLA 1999)*, Nov. 1999.
- [27] S. Letovsky, E. Soloway, "Delocalized Plans and Program Comprehension," *IEEE Software*, 3(3):41-9, 1986.
- [28] M. Lippert, C. V. Lopes, "A study on exception detection and handling using aspect-oriented programming," in *Intl. Conf. on Software Eng. (ICSE 2000)*, 2000, pp. 418-427.
- [29] A. Mockus, P. Zhang, P. Li, "Drivers for customer perceived software quality," in *International Conference on Software Engineering (ICSE 2005)*, 2005, pp. 225-233.
- [30] N. Nagappan, T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density," in *Intl. Conf. on Software Engineering (ICSE 2005)*, May 15-21 2005.
- [31] T. Ostrand, E. Weyuker, R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Transactions on Software Eng.*, 31(4):340-355, 2005.
- [32] M. Revelle, T. Broadbent, D. Coppit, "Understanding Concerns in Software: Insights Gained from Two Case Studies," in *International Workshop on Program Comprehension (IWPC 2005)*, May 15-16 2005.
- [33] M. P. Robillard, W. Coelho, G. C. Murphy, "How Effective Developers Investigate Source Code: An Exploratory Study," *IEEE Transactions on Software Engineering*, 30(12):889-903, December 2004.
- [34] M. P. Robillard, F. Weigand-Warr, "ConcernMapper: Simple View-Based Separation of Scattered Concerns," in *Wkshp. on Eclipse Tech. eXchange (ETX 2005)*, Oct. 2005.
- [35] J. Śliwerski, T. Zimmermann, A. Zeller, "When Do Changes Induce Fixes?," in *Workshop on Mining Software Repositories (MSR 2005)*, 2005, pp. 24-28.
- [36] S. L. Tsang, S. Clarke, E. Baniassad, "An Evaluation of Aspect-Oriented Programming for Java-based Real-time Systems Development," in *Intl. Symp. on OO Real-Time Distributed Computing (ISORC 2004)*, May 12-14 2004.
- [37] W. E. Wong, S. S. Gokhale, J. R. Horgan, "Quantifying the closeness between program components and features," *Journal of Systems and Software*, 54(2):87-98, 2000.
- [38] C. Zhang, H.-A. Jacobsen, "Quantifying Aspects in Middleware Platforms," in *Aspect-Oriented Software Development (AOSD 2003)*, March 2003, pp. 130-139.
- [39] T. Zimmermann, P. Weißgerber, S. Diehl, A. Zeller, "Mining Version Histories to Guide Software Changes," *IEEE Transactions on Software Eng.*, 31(6):429-445, 2005.