Do Crosscutting Concerns Cause Defects?

Marc Eaddy, *Student Member*, *IEEE*, Thomas Zimmermann, *Student Member*, *IEEE*, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, *Member*, *IEEE Computer Society*, Nachiappan Nagappan, *Member*, *IEEE*, and Alfred V. Aho, *Fellow*, *IEEE*

Abstract—There is a growing consensus that crosscutting concerns harm code quality. An example of a crosscutting concern is a functional requirement whose implementation is distributed across multiple software modules. We asked the question, "How much does the amount that a concern is crosscutting affect the number of defects in a program?" We conducted three extensive case studies to help answer this question. All three studies revealed a moderate to strong statistically significant correlation between the degree of scattering and the number of defects. This paper describes the experimental framework we developed to conduct the studies, the metrics we adopted and developed to measure the degree of scattering, the studies we performed, the efforts we undertook to remove experimental and other biases, and the results we obtained. In the process, we have formulated a theory that explains why increased scattering might lead to increased defects.

Index Terms—Crosscutting concerns, fault proneness, feature location, requirements traceability, mining software repositories, metrics, statistical analysis, empirical software engineering, open source software.

1 Introduction

Despite the significant effort that developers put into producing reliable software, defects still surface after the software is deployed. Defects creep in at every stage of the development process, avoid detection during testing, and all too often appear as failures to the user. Enormous effort goes into avoiding defects (e.g., defensive programming) and, when that fails, detecting defects (e.g., code inspections, program analysis, prerelease testing) to reduce the number of defects in a delivered software system. These efforts might be better directed if we had a better understanding of what causes defects.

This paper considers the possibility that one cause of defects is poor modularization of the concerns of the program. A *concern* is *any consideration that can impact the implementation of a program* [53]. A software requirement is an example of a kind of concern. When a concern's implementation is not modularized, that is, the implementation is scattered across the program and possibly tangled with the source code related to other concerns, the concern is said to be *crosscutting* [42]. Several empirical studies [25], [28], [29], [30], [47], [60], [64] provide evidence that crosscutting concerns degrade code quality because they negatively impact *internal quality metrics* (i.e., measures derived from the program itself [41]), such as program size, coupling, and separation of concerns.

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 [35], [38], such as maintenance effort, field reliability, and observed defects [21].

We argue in this paper that crosscutting concerns[1] might negatively impact at least one external quality attribute—*defects*, i.e., mistakes in the program text. Our theory is that a crosscutting concern is harder to implement and change consistently because multiple—possibly unrelated—locations in the code have to be found and 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. We hypothesize that this increased complexity leads to increased defects.

To formulate our theory, we present a formal model of concerns and their relationship to program elements and we introduce a set of metrics that measure the extent to which that relationship is crosscutting. To test our hypothesis, we conducted three case studies to gather data on scattering and defect counts. We then applied correlation analysis to gather empirical evidence of a *cause-effect relationship* between scattering and defects.

We found a moderate to strong correlation between scattering and defects for all three case studies. This suggests that scattering may cause or contribute to defects, which—if true—has many implications. First and foremost, our evidence suggests that one way we can improve software reliability is to modularize crosscutting concerns —or at least ensure they are well tested. Second, our

• *M. Eaddy, V. Garg, and A.V. Aho are with the Department of Computer Science, Columbia University, 1214 Amsterdam Avenue, New York, NY 10027. E-mail: {eaddy, vgarg, aho}@cs.columbia.edu.*
• *T. Zimmermann is with the Department of Computer Science, University of Calgary, 2500 University Drive NW, Calgary, AB, Canada T2N 1N4. E-mail: zimmerth@cpsc.ucalgary.ca.*
• *K.D. Sherwood and G.C. Murphy are with the Department of Computer Science, University of British Columbia, 201-2366 Main Mall, Vancouver, BC, Canada V6T 1Z4. E-mail: ducky@webfoot.com, murphy@cs.ubc.ca.*
• *N. Nagappan is with Microsoft Research, Software Reliability Research, One Microsoft Way, Redmond, WA 98052. E-mail: nachin@microsoft.com.*

Manuscript received 24 Sept. 2007; revised 15 Jan. 2008; accepted 12 Feb. 2008; published online 16 May 2008.
Recommended for acceptance by H. Ossher.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2007-09-0274.
Digital Object Identifier no. 10.1109/TSE.2008.36.

1. For this paper, we consider a crosscutting concern to be synonymous with a scattered concern [26].

findings suggest that cognitive complexity measures (e.g., concern-oriented metrics) are perhaps more appropriate predictors of software quality than structural complexity measures (e.g., coupling, code churn). Third, it prompts the need for independent replication of our results to build confidence that the relationship between scattering and defects is real. Finally, our findings call for additional research to determine the root cause of the supposed relationship: Are changes to highly crosscutting concerns more likely to be applied inconsistently? Are crosscutting concerns inherently more difficult to understand?

This paper proceeds as follows: In Section 2, we present a theory of the relationship between crosscutting and defects and state our research hypothesis. In Section 3, we describe our model of concerns and our suite of concern metrics that are based on the model. In Section 4, we outline the methodology we used to validate our theory. In Section 5, we describe our case studies. We present the results of our studies and a discussion in Section 6. We address threats to internal and external validities in Sections 7 and 8. We summarize related research in Section 9. Section 10 concludes.

2 Why Might Crosscutting Concerns Cause Defects?

Our theory is a set of models [31] that formalizes concepts such as "concerns," "program elements," and "defects," and describes their interrelationships, along with how they relate to the developer. In this section, we model the relationship between developers and concerns. We use the model to justify why crosscutting might cause defects, which we need to draw meaningful conclusions from our results [21].

Every line of code exists to satisfy some concern. Concerns may be described in many ways and at various levels of abstraction:

- Features from a feature list.
- Requirements from a software requirement document.
- Design patterns and design elements from a UML design document.
- Low-level programming concerns such as programming language used, coding style, programming idioms, code reuse, information hiding, and algorithms.

When faced with the task of implementing a concern, a developer creates—perhaps without realizing it—a *concern implementation plan* that guides her implementation decisions. It is in this plan that crosscutting first emerges. One developer's plan may entail scattering the implementation (e.g., she plans to copy-and-paste code), whereas another may localize it (e.g., she plans to create a shared function). The plan chosen depends on many variables, including the development process (e.g., priorities, time, resources), programming technology (e.g., program language), and the developer's aptitude.

The relationship between the concerns and the program is rarely documented [44]. This makes it difficult for maintainers of the program to answers questions such as *"Where are all the places that the* undo *feature is implemented?"* (i.e., top-down analysis [48]) and *"What is this piece of code for?"* (i.e., bottom-up analysis [48]). Without a proper understanding of the scattered nature of the concern implementation, maintainers may make changes incorrectly or neglect to make changes in all the right places.

Our conjecture is that, when the implementation of a concern is distributed (scattered) across many program elements, the complexity of that implementation increases, as does the difficulty of making changes correctly and consistently, increasing the likelihood of defects. Stated simply, crosscutting concerns are hard to find, understand, and work with. More formally, our research hypothesis is given as follows:

Hypothesis. *The more scattered a concern's implementation is, the more defects it will have, regardless of the implementation size.*

The last stipulation about size is necessary since past research has established that size, in terms of lines of code, is already a strong predictor of defects [22]. Since we expect scattering to be related to size, we must rule out the possibility that an increase in defects is caused by an increase in size alone. We will revisit this technicality in Section 6.3—so, for now, we ask the reader to ignore it.

Some controlled experiments on program understanding 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, resulting in several kinds of incorrect modifications [46]. Similarly, Robillard et al. 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" [54].*

Other studies indicate that programmers make mistakes when modifying classes whose implementations are scattered due to inheritance. Harrison et al. found that *"systems without inheritance are easier to modify than systems with either three or five levels of inheritance"* [32]. From the perspective of our theory, inheritance scatters the implementations of the underlying concerns.

In another study, Bruntink et al. observed that the idiom used to implement a specific crosscutting concern (exception handling) made it *"too easy to make small mistakes [that] can lead to many faults spread all over the system"* [9].

Finally, enhancements or 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 [49] and we propose that changes to crosscutting concerns may be the root cause.

To validate our theory empirically and test our hypothesis, we next describe our concern model and a suite of metrics that operationalize the concept of "highly scattered."

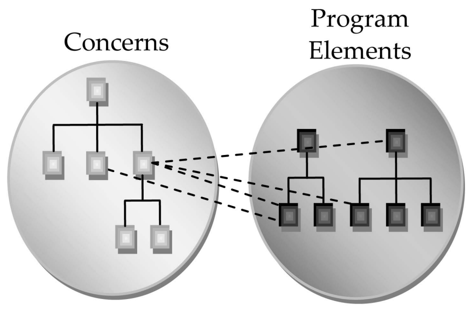


Fig. 1. Relation between concerns and program elements.

3 A Model of Concerns

Abstractly, a *program specification*, or simply *specification*, is a description of a program. A specification may be *executable*, e.g., a set of program elements, or *nonexecutable*, e.g., a requirements specification or architectural design. Our operational definition of a concern is *an item from a program's nonexecutable specification.* Thus, a nonexecutable specification represents a *concern domain* of the program.

We define our *concern-program element mapping* as a tuple $M = (S, T, C_S, C_T, R)$. S is a set of concerns organized into a hierarchy [59] described by $C_S = \{(s_1, s_2) | s_1, s_2 \in S, s_1 \neq s_2, s_1 \text{ is the parent of } s_2\}$. T is a set of program elements organized according to $C_T = \{(t_1, t_2) | t_1, t_2 \in T, t_1 \neq t_2, t_1$ is the parent of t_2 in the abstract syntax tree [1] of the program$\}$. Finally, R is the relation of interest between the two specifications, $R = \{(s, t) | s \in S, t \in T\}$. This is depicted in Fig. 1.

Note that C_T does not describe a class inheritance hierarchy. It describes a forest of trees, the roots of which are the abstract syntax trees of the individual source files, which syntactically contain class definitions, which in turn contain class member definitions, and so forth.

The program elements that are meaningful depend upon the language in which the program is expressed. The projects analyzed in this paper were written in Java, so we are primarily interested in classes, fields, methods, and statements.

3.1 Terminology

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 [5], [17], [24] . For the purposes of this paper, *a crosscutting concern is a concern that is scattered* [26, p. 4].

This binary definition of scattering is simple and unambiguous but is not very useful when most of the concerns are scattered, which we believe to be the rule rather than the exception [17], [62]. Hence, we need metrics to determine the degree of scattering (DOS).

3.2 Concern Metrics

There are many ways to describe how a concern is implemented. For the purpose of validating our theory, we focused on four cognitive complexity metrics that describe how scattered the concern's implementation is, in absolute terms and in terms of statistical distribution, and with respect to classes and methods (the elements of interest in an object-oriented implementation). This allows us to determine which characteristic of scattering, if any, is the best predictor of defects.

Complexity metrics tend to be heavily influenced by size (in terms of lines of code), which can lead a researcher to perceive a cause-effect relationship where none exists [22]. To test for a possible influence, we also measured the concern's size, i.e., the total number of lines of code associated with the concern. We discuss the results of the concern size tests in Section 6.3.

Table 1 provides a summary of the metrics, which we will now describe in detail.

3.2.1 Program Element Contribution

Program element contribution (CONT) is the number of lines of code in a program element that are associated with a concern. The entire line is counted even if only a portion is associated with the concern. Indeed, a line may be associated with multiple concerns.

For a method or field associated with a concern, the contribution is the number of lines in the method (method declaration plus method body) or field declaration.

For classes, the contribution includes the lines of the class declaration plus the contributions of the class's

TABLE 1
Concern Metrics

Bug Count	Number of unique bugs associated with the concern.
Program Element Contribution (CONT)	Number of lines of code in the program element that are associated with the concern. In general, whitespace or comments are excluded; however, for one case study, they were included. Lines outside of class definitions (e.g., package declaration, imports) are not counted by our tool.
Lines of Concern Code (LOCC)	Total number of lines of code that contribute to the implementation of a concern.
Concern Diffusion over Components (CDC)	Number of classes that contribute to the implementation of a concern and other classes and aspects which access them. [28]
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. [28]
Degree of Scattering across Classes (DOSC)	Degree to which the concern code is distributed across classes. Varies from 0 to 1. When DOSC is 0 all the code is in one class. When DOSC is 1 the code is equally divided among all the classes. [17]
Degree of Scattering across Methods (DOSM)	Degree to which the concern code is distributed across methods. Varies from 0 to 1 similar to DOSC. [17]

methods and fields. *Inner classes* in Java are considered separate from the enclosing class when determining contribution and *anonymous classes* are considered part of the enclosing method. Note that inheritance has no bearing on contribution.

When the element is the entire program, P, the contribution is the sum of the contributions of all the classes, i.e., the total number of lines associated with the concern s. We give this special case its own metric, *lines of concern code (LOCC)*, that is, $\mathrm{LOCC}(s) = \mathrm{CONT}(s, P)$.

3.2.2 Scattering Metrics

The *concern diffusion metrics*, created by Filho et al., measure scattering in absolute terms as a count of the number of classes (CDC) or methods (CDO) that implement the concern [25]. We include CDC and CDO in our correlation analysis because they are rigorously defined, are validated by several studies [25], [28], [30], and they nicely contrast our degree of scattering metric.

The *degree of scattering metric* created by Eaddy et al. [17] provides more information by further considering how the concern's code is distributed among the elements. We believe this 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. The degreee of scattering metric builds upon the *concentration metric* (CONC) introduced by Wong et al. [62]:

$$\mathrm{CONC}(s,t) = \frac{\textit{Source lines in element } t \textit{ related to concern } s}{\textit{Source lines related to concern } s} \tag{1}$$

$$= \frac{\mathrm{CONT}(s,t)}{\mathrm{CONT}(s,P)}. \tag{2}$$

For the object-oriented programs we studied, we measured *degree of scattering across classes* (DOSC), in which case t is a class, and *degree of scattering across methods* (DOSM), in which case t is a method.

Degree of scattering is a measure of the *statistical variance* [37, p. 57] of the concentration of a concern over all program elements with respect to the worst case (i.e., when the concern is equally scattered across all elements):

$$\mathrm{DOS}(s) = 1 - \frac{Variance(s)}{Variance_{ideal}(s)}, \tag{3}$$

where

$$Variance(s) = \frac{\sum_{t \in T} (\mathrm{CONC}(s,t) - \mathrm{CONC}_{worst})^2}{|T|}. \tag{4}$$

The worst case occurs when the implementation of a concern is uniformly distributed across all program elements in T, i.e., $\mathrm{CONC}_{worst} = 1/|\mathrm{T}|$. Substituting this into (4),

$$Variance(s) = \frac{\sum_{t \in T} \left(\mathrm{CONC}(s,t) - \frac{1}{|T|}\right)^2}{|T|}. \tag{5}$$

The ideal variance occurs when CONC is 1 for one component t and 0 for all other components, i.e., the concern s is completely localized in t. Equation (5) reduces to

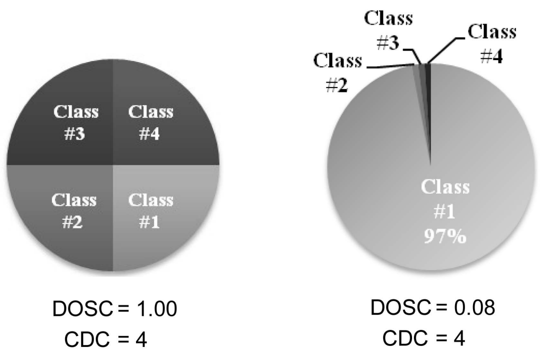


Fig. 2. Comparing DOSC and CDC for two different implementations of the same concern.

$$Variance_{ideal}(s) = \frac{|T| - 1}{|T|^2}. \tag{6}$$

Substituting (6) into (3) and simplifying,

$$\mathrm{DOS}(s) = 1 - \frac{|T| \sum_{t \in T} \left(\mathrm{CONC}(s,t) - \frac{1}{|T|}\right)^2}{|T| - 1}. \tag{7}$$

Using the validation methodology and terminology specified by Kitchenham et al. [43], DOS, and by extension DOSC and DOSM, has the following properties:

- It is normalized to be between 0 (*completely localized*) and 1 (*completely delocalized; uniformly distributed*) (inclusive) so that concerns can be meaningfully compared. DOS can theoretically take on any real value within this range and is therefore *continuous*. DOS is undefined when $|T| \leq 1$.
- DOS is *proportional* to the number of elements related to the concern and *inversely proportional* to the concentration. That is, the less concentrated the concern is, the more scattered it is.
- DOS is a *ratio-scale measure* (0 means "no scattering"). Thus, it is meaningful to compare and rank concerns by their DOS values and obtain the average DOS.
- While DOS is unitless, the individual components of the DOS equation do have units, specifically, the units are *lines of code (LOCs)*, T, and the structural unit of T (e.g., classes, methods). One can directly compare two DOS values only if they are both obtained from DOS equations with identical units. This implies that it is not meaningful to directly compare DOS values for two different programs or two different versions of the same program when S or T is different.

3.2.3 Comparing DOSC and CDC

The difference between DOSC and CDC is illustrated in Fig. 2. The pie charts show how the 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). In the second, the implementation is mostly localized. We compute DOSC as follows:

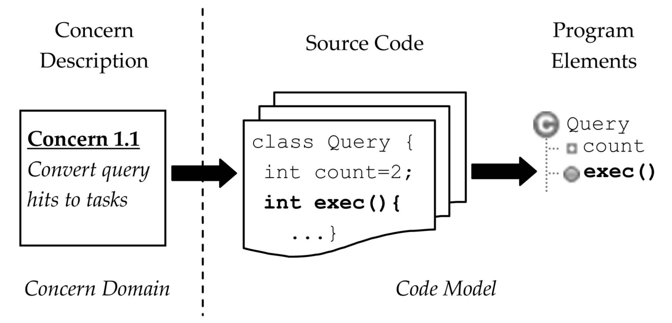


Fig. 3. Associating concerns with program elements.

$$\mathrm{DOSC} = 1 - \frac{|4|\left(\left(0.25 - \frac{1}{|4|}\right)^2 + \left(0.25 - \frac{1}{|4|}\right)^2 + \left(0.25 - \frac{1}{|4|}\right)^2\right)}{|4| - 1}$$
$$= 1.$$

In the second scenario, the DOSC value is

$$\mathrm{DOSC} = 1 - \frac{|4|\left(\left(0.97 - \frac{1}{|4|}\right)^2 + \left(0.01 - \frac{1}{|4|}\right)^2 + \left(0.01 - \frac{1}{|4|}\right)^2 + \left(0.01 - \frac{1}{|4|}\right)^2\right)}{|4| - 1}$$
$$= 0.08.$$

DOSC is close to 0, indicating the implementation is mostly localized. CDC cannot distinguish the two implementations, as evident by the value of 4 for both.

4 Methodology Used to Validate Our Theory

To validate our theory, we chose to undertake a series of case studies of open source Java programs. In particular, we looked for medium-sized programs that had a clear set of software requirements and documented defects (the reasons for these criteria will become apparent in a moment). For the three programs we selected, we reverse engineered the concern-code mapping and the bug-code mapping. We then inferred the bug-concern mapping. After obtaining the three mappings, we were able to compute the metrics described in the previous section and measure the correlation between scattering and defects.

More formally, our methodology for obtaining the mappings consists of the following steps:

1. **Reverse engineer** the *concern-code mapping:* S and C_S (Section 4.1), and R (Section 4.2). This part of our methodology, depicted in Fig. 3, is subjective. However, we defined assignment rules to improve the repeatability of our mappings and chose statistical methods designed to improve the reliability of our correlation results.
2. **Mine** the *bug-code mapping:* S is the set of bugs and R is automatically determined using bug fix data. This is depicted in Fig. 5 and described in Section 4.4.
3. **Infer** the *bug-concern mapping.* Section 4.5 explains how we associate a bug with a concern if the concern's implementation was modified to fix the bug (depicted in Fig. 6).

4.1 Concern Selection

Selecting the right set of concerns to analyze is critical to ensure that our theory is applicable, our statistical analysis is valid, and our results are meaningful. However, our broad definitions for "concern" and "nonexecutable specification" imply an infinite number of concerns from which to choose. The context of our theory reduces the scope to *actual* concerns, i.e., there is evidence that the concerns provide the rationale for the implementation. For example, *maintainability* is not an actual concern if the developer did not consider it. This is important because our theory only explains defects when they are related to actual crosscutting concerns. This requirement was difficult to satisfy as most of the 75 open source projects[2] we considered did not have requirement documents.

Another criterion was that the set of concerns should provide a rationale for most of the code. This reduces *sample bias* since all concerns are considered, not just those that are crosscutting. Furthermore, to ensure that our correlations were statistically significant, we required that the final concern set include at least 30 concerns [38, p. 64]. This is easily accomplished by making concerns more granular; however, at some point, we must increase the granularity of the program elements assignable to the concerns or suffer a loss in precision. For example, associating a concern with an entire method when it is only related to a single statement inflates the concern's size. Unfortunately, our concern and bug assignment tools, and time restrictions, limited us to field and method-level granularity (e.g., we could not assign individual statements). We discuss how this limitation affects internal validity in Section 7.3.

The actual process of selecting concerns involved determining 1) the appropriate concern domain (e.g., the software requirement specification), 2) what constitutes a concern in that domain, including the concern granularity, and 3) the concern hierarchy. The final concern hierarchy is entered into a tool we built, called ConcernTagger, so that we may begin assigning program elements to the concerns. We give examples of concerns for the three case studies in Sections 5.1, 5.2, and 5.3. We describe the tool and assignment procedure in Section 4.2.

4.2 Concern Assignment

Concern assignment is the process of determining the relationship between a concern and a program element [6]. In our methodology, an *analyst* determines the relationship by examining a set of concern descriptions and the source code (see Fig. 3). For our studies, the most relevant relationship between concerns and program elements would be based on a *likely-to-contain-defect rule:*

A program element is relevant to a concern if it is likely to harbor defects related to that concern.

In other words, if a bug is reported for a concern, the defect is likely to lie in one of these program elements. Obviously,

2. The list is available in the Online Appendix, which can be found at http://www.cs.columbia.edu/~eaddy/concerntagger.

this relationship is difficult, perhaps impossible, to establish with any certainty. Instead, we approximate this rule using the *prune dependency rule* created by Eaddy et al. [17], which is easier to decide:

A program element is relevant to a concern if it should be removed, or otherwise altered, when the concern is pruned.

To properly interpret this rule, consider a *software pruning* scenario where a developer is removing a concern to reduce the footprint of a program or otherwise tailor the program for a particular environment. In this case, they want to remove as much code as possible, short of a redesign,[3] and without affecting other concerns.

A benefit of the prune dependency rule is that the mapping can be directly obtained by actually removing each concern in turn and noting which elements require changes. However, this task is very labor-prone and was not feasible for the scale of the projects we studied. We therefore relied on a human analyst to estimate the mapping. Based on our experience assigning the concerns of five small to medium-sized projects (13-44 KLOCs) by hand, we believe a prune dependency is easier to estimate than other types of relationships (e.g., *implements* [50], *contributes-to* [25], [52]) and produces relevant results [17]. In this context, *relevance* is the extent to which the prune dependency mapping agrees with the likely-to-contain-defect mapping.[4] Both rules will exclude "obviously" irrelevant program elements, including methods shared by all concerns (e.g., the *main* function), general purpose methods (e.g., *String.concat*), and elements contained in system and generic libraries.[5] On the other hand, a prune dependency assignment will include some elements that are unlikely to contain defects, e.g., field declarations and accessor methods.

Deciding if a prune dependency relationship exists requires human judgment and is therefore subject to human error. Fortunately, our statistical analysis method (Spearman's correlation) mitigates the impact of these measurement errors since it only considers the relative ordering of values, not the absolute values themselves. We revisit the issue of assignment error in Section 7.1.

The actual assignment of elements to concerns was done by two of the authors using an extension to ConcernMapper [55], a plug-in for the Eclipse[6] development environment, developed by Robillard et al. ConcernMapper allows the user to associate program elements with concerns via drag-and-drop and so forth. Our extension to ConcernMapper, named ConcernTagger,[7] further allows the user to create a hierarchy of concerns and obtain concern metrics and assignment coverage statistics (see Fig. 4).

The analyst carries out the concern assignment task by systematically inspecting each program element and deciding if the prune dependency rule applies to any of the concerns. In some cases, this decision is easy, e.g., any field named "log" has a prune dependency on the *logging* concern. However, we found that the accuracy of the majority of the decisions hinged on how well the analyst understood the program. To aid program understanding, we relied on project documentation, source code comments, code navigation and search tools, change history comments, and, in the case of the Rhino study, unit tests.

3. Assume that disabling the concern using a flag, preprocessor macros, or code generation is not allowed.

4. If we knew the likely-to-contain-defect mapping, we would measure similarity directly using the Jaccard similarity metric [58], for example.

5. It is not necessary to consider elements contained in system and generic libraries because application-specific concerns generally do not provide a rationale for general-purpose code (a similar argument is made in [19]).

6. http://www.eclipse.org.

7. http://www.cs.columbia.edu/~eaddy/concerntagger.

4.3 Ensuring Independence of Concern Metrics

Correlation and regression analysis can only be applied to concerns whose concern metrics are *independent* [37, pp. 114, 206]. As we mentioned, concerns may be organized in a containment hierarchy, in which case the observation below applies.

Observation. *The program elements associated with a concern via the prune dependency rule must (at least) include the program elements associated with the concern's descendants.*

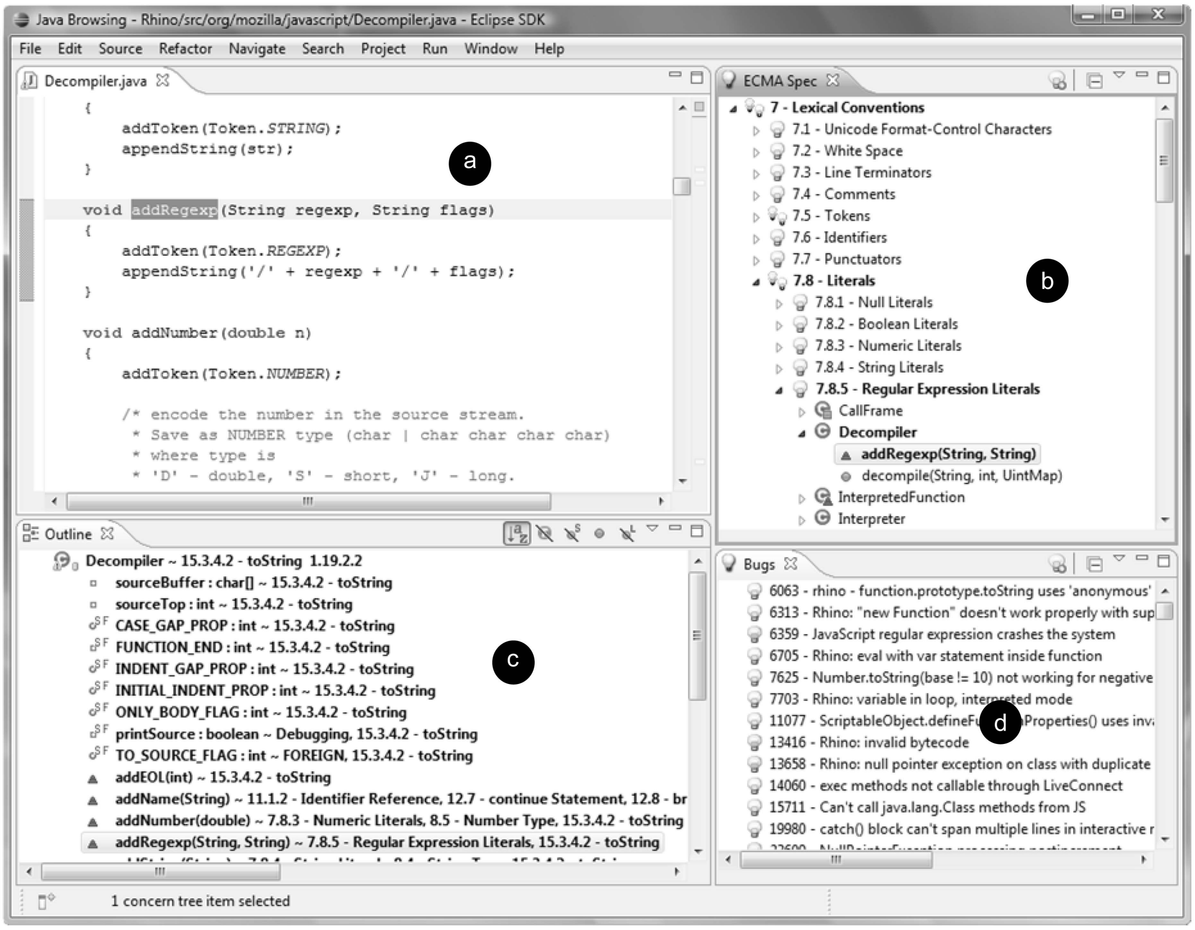
Justification. The prune dependency assignment rule states that a program element is associated with a concern if removing the concern would require modification or removal of the element. Therefore, when concerns are organized in a containment hierarchy, removing a parent concern implies that the parent's descendants are also removed. Since removing the parent's descendants requires modification or removal of the program elements associated with the descendants, it follows that the parent concern must also be associated with those elements.

Our concern metrics are derived from the program elements associated with a concern. The observation above implies that the concern metrics for a parent concern are dependent on those of its descendant concerns (i.e., the metrics are *collinear*). For example, the *root concern* has the largest size and bug count and is the most scattered.[8] Correlation and regression analysis is undefined when the metrics of the concerns are interdependent [22]. Therefore, although we assigned all of the concerns, we only performed statistical analysis on sets of concerns where no two concerns were descendants of each other (specifically, *leaf concerns*). Restricting our analysis in this way does not introduce sample bias since the leaf concerns provide the rationale for most of the code, as our concern coverage statistics (discussed in Section 6) show.

4.4 Bug Assignment

As is typical, we did not have records of individual defects. Instead, we relied on records of *bugs: bug reports* stored in an *issue tracking system* (ITS) and *bug fixes* stored in a *source code control system* (SCCS) [56]. A *bug* is caused by one or more defects. For example, a user might report a crash (i.e., a *failure* [34]) that is caused by multiple defects, whereas a developer might report access to an uninitialized variable (a single defect). To validate our theory, we approximate

8. Except in rare cases, the DOS metrics for a parent will be greater than or equal to its children.


Java Browsing - Rhino/src/org/mozilla/javascript/Decompiler.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Decompiler.java
ECMA Spec
Outline
Bugs
a
b
c
d

```
    {
        addToken(Token.STRING);
        appendString(str);
    }

    void addRegexp(String regexp, String flags)
    {
        addToken(Token.REGEXP);
        appendString('/' + regexp + '/' + flags);
    }

    void addNumber(double n)
    {
        addToken(Token.NUMBER);

        /* encode the number in the source stream.
         * Save as NUMBER type (char | char char char char)
         * where type is
         * 'D' - double, 'S' - short, 'J' - long.
```

```
7 - Lexical Conventions
  7.1 - Unicode Format-Control Characters
  7.2 - White Space
  7.3 - Line Terminators
  7.4 - Comments
  7.5 - Tokens
  7.6 - Identifiers
  7.7 - Punctuators
  7.8 - Literals
    7.8.1 - Null Literals
    7.8.2 - Boolean Literals
    7.8.3 - Numeric Literals
    7.8.4 - String Literals
    7.8.5 - Regular Expression Literals
      CallFrame
      Decompiler
        addRegexp(String, String)
        decompile(String, int, UintMap)
      InterpretedFunction
      Interpreter
```

```
Decompiler ~ 15.3.4.2 - toString 1.19.2.2
    sourceBuffer : char[] ~ 15.3.4.2 - toString
    sourceTop : int ~ 15.3.4.2 - toString
    CASE_GAP_PROP : int ~ 15.3.4.2 - toString
    FUNCTION_END : int ~ 15.3.4.2 - toString
    INDENT_GAP_PROP : int ~ 15.3.4.2 - toString
    INITIAL_INDENT_PROP : int ~ 15.3.4.2 - toString
    ONLY_BODY_FLAG : int ~ 15.3.4.2 - toString
    printSource : boolean ~ Debugging, 15.3.4.2 - toString
    TO_SOURCE_FLAG : int ~ FOREIGN, 15.3.4.2 - toString
    addEOL(int) ~ 15.3.4.2 - toString
    addName(String) ~ 11.1.2 - Identifier Reference, 12.7 - continue Statement, 12.8 - br
    addNumber(double) ~ 7.8.3 - Numeric Literals, 8.5 - Number Type, 15.3.4.2 - toString
    addRegexp(String, String) ~ 7.8.5 - Regular Expression Literals, 15.3.4.2 - toString
```

```
6063 - rhino - function.prototype.toString uses 'anonymous'
6313 - Rhino: "new Function" doesn't work properly with sup
6359 - JavaScript regular expression crashes the system
6705 - Rhino: eval with var statement inside function
7625 - Number.toString(base != 10) not working for negative
7703 - Rhino: variable in loop, interpreted mode
11077 - ScriptableObject.defineFunctionProperties() uses inv
13416 - Rhino: invalid bytecode
13658 - Rhino: null pointer exception on class with duplicate
14060 - exec methods not callable through LiveConnect
15711 - Can't call java.lang.Class methods from JS
19980 - catch() block can't span multiple lines in interactive r
```

1 concern tree item selected

Fig. 4. ConcernTagger screenshot showing (a) a Rhino source file, (b) the Rhino concern hierarchy showing the program elements assigned to the "Regular Expression Literals" concern (program elements can be assigned to concerns via drag-and-drop and right click), (c) a view showing which concerns are assigned to the methods of the Decompiler class, and (d) the Rhino bugs.

defect counts, which are not directly measurable, with *bug counts*, which are directly measurable, as we will soon see.

4.4.1 Associating Bugs with Bug Fixes

When a bug report is filed in the ITS, the bug is given a unique *bug ID.* The open source projects we analyzed had publicly accessible issue tracking systems, so the filer could be a developer on the project or a user (or both). If the bug is genuine, not a duplicate, and is caused by defects in one or more source files, a developer eventually fixes it, submits the updated files to SCCS along with a reason for the changes, and then changes the bug status to "fixed." We use the term *bug fix* to refer to the set of lines in the source code—which may span multiple files—added, removed, or modified to fix a bug.

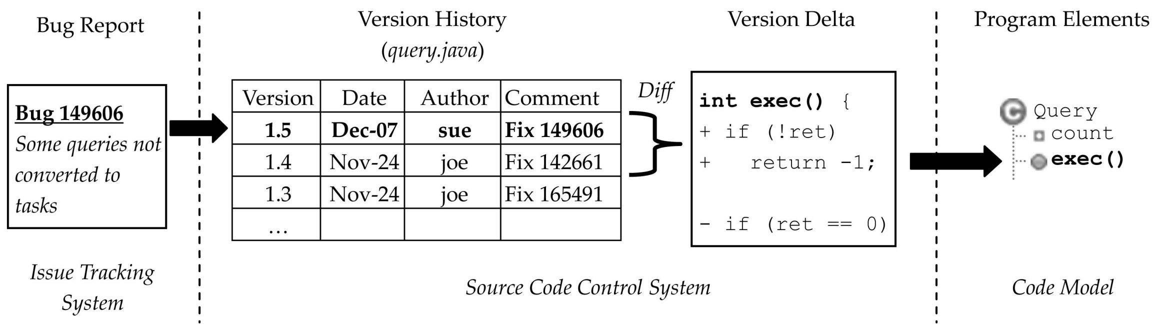
Common SCCSs typically record the changes made to source files in the form of one or more *deltas.* A delta provides a list of the lines added, removed, and modified and the reason for the change (called the *commit message*). The SCCS systems used by the projects we studied were CVS [12] and Subversion [15]. For CVS, the *unit of change* described by a delta is a single file, so a fix may consist of multiple deltas. For Subversion, the unit of change can include multiple files, so a fix consists of one delta.

A common approach for associating bugs with program elements is to search for deltas whose commit messages include keywords such as "bug" or "fix" [51] or include strings that look like bug IDs [16], [27], [57]. However, relying on this information alone is insufficient. For one project we studied, the IDs in 87 (37 percent) of the commit messages referred to enhancements instead of bugs, which would have inflated the bug counts for some concerns. This is easily prevented by using the issue tracking system to verify that IDs refer to actual bugs. Of course, bugs identified by keywords instead of IDs cannot be systematically verified using this approach.

Furthermore, it is common for a bug to be fixed incorrectly the first time [51] or be worked on in stages, requiring multiple updates to the same file [2]. This can result in the same bug being counted multiple times. Again, using bug IDs helps us minimize noise since we only count unique bug IDs.

Our approach for recognizing bug fixes is depicted in Fig. 5 and described in detail by Śliwerski et al. [57], which is similar to the approaches used by Fischer et al. [27] and by Čubranić et al. [16]:

> *A delta is called a "bug fix" and associated with a bug if the change reason refers to a valid bug ID according to the issue tracking system.*



Version	Date	Author	Comment
1.5	**Dec-07**	**sue**	**Fix 149606**
1.4	Nov-24	joe	Fix 142661
1.3	Nov-24	joe	Fix 165491
...			

```
int exec() {
+ if (!ret)
+    return -1;

- if (ret == 0)
```

Fig. 5. Associating bugs with program elements.

For Bugzilla, a valid bug is an issue in the ITS with a *resolution* of "fixed," a *status* of "closed," "resolved," or "verified," and a *severity* that is not "enhancement." For Jira, the *type* must be "bug," the *resolution* must be "fixed," and the *status* must be "closed" or "resolved." We included bug fixes associated with any branch in the version database (not just the main branch).

Here are examples of commit messages from the projects we studied that our approach associates with a bug:

"NEW—bug 172515: Synchronizing queries with Bugzilla stuck when empty results. https://bugs.eclipse.org/bugs/show_bug.cgi?id=172515"

"Fix for 305323: Rhino fails to select the appropriate overloaded method."

"Fix for JIRA IBATIS-260: "Hash conflict with groupBy resultMaps""

We required that the majority of bugs in the ITS be traceable to bug fixes using this approach. This helps ensure that we do not miss bugs that should be assigned to program elements (false negatives) and that our correlation results are statistically significant. This turned out to be a very stringent requirement. Out of the 75 medium-sized (less than 50 KLOCs) open source projects we considered for our case studies, very few followed the practice of including bug IDs in commit messages. However, this requirement ensured that our defect counts would be sufficiently accurate for our purposes.

4.4.2 Associating Bugs with Program Elements

To decide if a bug is associated with a program element, we created the *fixed-for-bug rule*:

A program element is relevant to a bug if it was modified to fix the bug.

For the first case study (Mylyn-Bugzilla), the first author associated bugs with bug fixes and then program elements, by hand.[9] We realized that this procedure (depicted in Fig. 5) could be easily automated, which would eliminate inconsistencies caused by human error. We created a plug-in for Eclipse, named BugTagger, which automatically associates bugs from a Bugzilla or Jira issue tracking system with methods, fields, and types, using change history from a CVS or Subversion database.

9. To prevent bias, concern assignment was performed by a different author.

4.5 Automatically Assigning Bugs to Concerns

Once we have mapped concerns and bugs to program elements, it is trivial to automatically associate bugs with concerns:

A bug is associated with a concern if the bug occurs in the concern's implementation, i.e., the intersection of the sets of program elements associated with the bug and the concern is nonempty.

This is depicted in Fig. 6. Our underlying assumption is that it is reasonable to associate a bug with a concern if the source code associated with the concern must be changed to fix the bug. This echoes the approach that is common in the software engineering literature (for example, see [22]), where a defect is assigned to a class if it occurs in the class's implementation. The bug count for a concern is therefore the number of *unique* bugs associated with the concern.

Our bug-concern assignment methodology does not consider the similarity of the sets of program elements assigned to the concern and bug, other than requiring that at least one element is shared. We therefore make no claims about the *strength* of the association between a bug and a concern. For example, if all of the program elements modified to fix a bug were associated with one concern, we would say that the bug was strongly associated with that concern. For the purposes of validating our theory, we only need to know how defective a concern is and, for this, our bug-concern assignment rule is adequate.

5 Our Case Studies

Case studies in software engineering test theories and collect data through observation of a project in an unmodified setting [63]. In this section, we summarize the programs we studied, explain how we selected the concerns, and provide some sample concerns.[10] We required all three projects to share the following characteristics:

- *Open source*—Ensures that our studies can be replicated. In addition, program understanding, which is required for concern assignment, is very difficult without access to the source code [6].
- *Written in Java*—Limitation imposed by our tooling.

10. Due to space limitations, we could not list all of the concerns we analyzed. However, the complete list is available in the Online Appendix, which can be found at http://www.cs.columbia.edu/~eaddy/concerntagger.

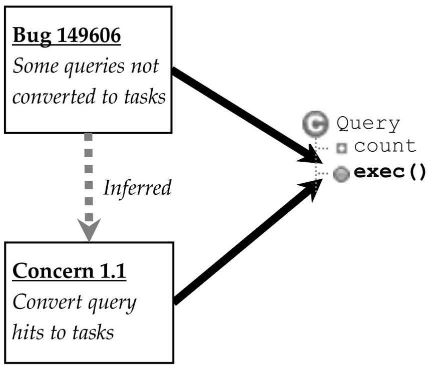

Bug 149606
Some queries not converted to tasks
Inferred
Concern 1.1
Convert query hits to tasks
Query
count
exec()

Fig. 6. Associating bugs with concerns.

- *Production quality*—Helps us to argue that our results generalize to an industrial setting.
- *Maintained by several people*—This is more representative of industrial projects. Furthermore, our theory has greater applicability to highly collaborative projects, where we suspect the adverse effects of scattering to be more evident.
- *Easily identifiable set of at least 30 relevant concerns* (see Section 4.1).
- *Publicly accessible ITS* (see Section 4.4.1).
- *Majority of bugs are referenced consistently by ID in commit messages* (see Section 4.4.1).

To improve the generalizability of our results, we purposely varied some context parameters, such as application domain and project size. Table 2 summarizes the high-level project differences.

5.1 Case Study 1: Mylyn-Bugzilla

Mylyn[11] is a production-quality plug-in for the Eclipse development environment that enables a task-focused development methodology [39]. It was developed by a team of graduate 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)[12] computed using the Unix *wc* command; however, we limited our analysis to two components: bugzilla.core and bugzilla.ui, totaling 56 classes, 427 methods, 457 fields, and 13,649 lines of Java code. We refer to this subset as Mylyn-Bugzilla.

The requirements for Mylyn-Bugzilla were reverse engineered based on the "New and Noteworthy" section of the Mylyn Web site and the personal experience of one of the authors with the development and usage of the components. We identified 28 of Mylyn's functional and nonfunctional requirements related to the bugzilla.core and bugzilla.ui components (i.e., requirement concerns). This is somewhat short of the 30 concern requirement we put forth in Section 4.1. We explain how this affected statistical significance in Section 6.1. The requirements were organized as a list so they were all leaf concerns. Examples of requirement concerns are "Convert query hits to tasks" and "Support search for duplicates."

11. http://www.eclipse.org/mylyn.
12. For Mylyn-Bugzilla, line counts include comments and whitespace (i.e., LOCs). They are excluded for the other two case studies so that only source lines are counted (i.e., SLOCs).

For Mylyn-Bugzilla, one author (heretofore referred to as "Author A") manually assigned concerns to program elements using the procedure outlined in Section 4.2. To avoid potential bias, a different author ("Author B") manually assigned bugs to program elements using the procedure explained in Section 4.4. As explained in Section 4.5, the assignment of bugs to concerns was completely automated for all the case studies.

5.2 Case Study 2: Rhino

Rhino[13] is a JavaScript/ECMAScript interpreter and compiler. Rhino began life as an industrial project at Netscape and was then transitioned to open source. Due to its large user base and extensive test suite, Rhino has a healthy number of bugs in its bug database. Version 1.5R6 consists of 32,134 source lines of Java code (SLOCs), 138 types (classes, interfaces, and enums), 1,870 methods, and 1,339 fields (as reported by ConcernTagger).

Unlike the other case studies, Rhino implements a formal specification: *the ECMAScript Standard* [18]. Obviously, this specification provides a strong rationale for at least part of the source code of any program that claims to *conform* to the specification. It was therefore an obvious choice for the concern domain. Every normative section and subsection of the specification was considered a concern, resulting in a hierarchy of 480 concerns. However, to ensure that our samples were independent (as explained in Section 4.3), we only performed statistical analysis on 357 mapped leaf concerns.

The screenshot of ConcernTagger in Fig. 4 shows a portion of the Rhino concern hierarchy. The "7 Lexical Conventions" concern is visible, along with its subconcern "7.8 Literals," which has the child leaf concern named "7.8.5 Regular Expression Literals." Also visible are some of the program elements assigned to the Regular Expression Literals concern, which would need to be modified or removed if support for regular expression literals was removed. Refer to [18] for detailed concern descriptions.

For Rhino, Author B manually assigned concerns to program elements, while BugTagger automatically assigned bugs as explained in Section 4.4.2.

5.3 Case Study 3: iBATIS

iBATIS[14] is a popular object-relational mapping (O/RM) tool for persisting Java objects in a relational database. The project was started by a single developer in 2001 and has since gathered a community of collaborators. The community currently includes 12 active developers, some with industrial experience. Version 2.3 consists of 13,314 source lines of Java code, 212 classes, 1,844 methods, and 536 fields (as reported by ConcernTagger).

The iBATIS Developer's Guide provides a good overview of functionality but makes for a poor concern domain. One reason is that the guide was clearly written after iBATIS was implemented—it is a stretch to say that the guide provides a rationale for the implementation. Concerns *cause* implementation, not the other way around, and therefore concerns must precede the implementation in

13. http://www.mozilla.org/rhino.
14. http://ibatis.apache.org.

TABLE 2
Project Summaries

	Mylyn–Bugzilla	*Rhino*	*iBATIS*
Application Domain	Development Tools	Compilers	Databases
Project Size (KLOCS)	Small (~14)	Medium (~32)	Small (~13)
Issue Tracking System	Bugzilla	Bugzilla	Jira
Source Code Control System	CVS	CVS	Subversion
Concern Domain	Requirements	ECMAScript Specification	Requirements
Bug Assignment Technique	By-Hand	BugTagger	BugTagger

TABLE 3
Size and Assignment Coverage Statistics

	Mylyn–Bugzilla			*Rhino*			*iBATIS*		
	All	Mapped[a]	%	All	Mapped	%	All	Mapped	%
Classes	56	44	79	138	80	57	212	207	97
Methods	427	253	59	1870	1415	75	1844	1807	97
Fields	457	230	50	1339	962	71	536	529	98
Lines[b]	13649	5914	43	32134	28308	88	13314	13144	98
Concerns[c]	28	28	100	480	417	86	183	173	94
Bugs[d]	110	101	92	241	160	66	87	47	53

[a] *A program element is "mapped" if it is assigned to at least one concern.*
[b] *Comments and whitespace are included for Mylyn-Bugzilla but excluded for Rhino and iBATIS.*
[c] *"All" means all concerns in the concern domain. "Mapped" means those concerns that were assigned to at least one program element.*
[d] *"All" means all "fixed" bugs (nonenhancement issues) found in the ITS on or before the version of the program we studied. "Mapped" means bugs*

time. Furthermore, the concepts in the guide are clearly organized and presented in a way that guides learning, not implementation. We therefore constructed a mock requirement document based on the guide consisting of 183 requirement concerns organized in a hierarchy. Of these concerns, 132 were leaves. An example of a requirement concern is "Caching," which has subconcerns "Class Caching," "Request Caching," and "Statement Caching."

For iBATIS, Author B manually assigned concerns, while BugTagger automatically assigned bugs.

6 Empirical Results and Discussion

Table 3 shows the amount of source code covered by the selected concerns and bugs. The concern coverage for Mylyn-Bugzilla is relatively poor, considering that only 43 percent of the code is covered by the requirement concerns we reverse engineered. This is likely due to a lack of a complete set of requirements for the Mylyn-Bugzilla component. In contrast, the bug coverage is high (92 percent).

The concern coverage for Rhino is high (88 percent), confirming that the ECMAScript specification explained most of the code. The remaining 12 percent is dead code, general purpose, or implements other concerns. For example, Rhino implements some nonstandard extensions to ECMAScript, as well as the E4X and LiveConnect standards. The bug coverage was somewhat low (66 percent), probably because some bugs were mapped to program elements that were absent in the version of Rhino we studied (1.5R6) or were related to concerns other than the ones we analyzed (e.g., LiveConnect).

Among studies that map concerns manually and exhaustively—as opposed to the more common approach of only mapping a subset of the concerns or a portion of the code—Rhino is the largest and most comprehensive study that we know of.

We obtained 98 percent concern coverage for iBATIS, signifying that the developer's guide we used to create the requirements described all the functionality. The bug coverage was somewhat low (53 percent), probably because of the issues already mentioned for Rhino. Low bug coverage is not necessarily bad, as we explain in Section 7.2.4.

6.1 Is Scattering Correlated with Defect Count?

Fig. 7 shows the scatter plots for all of the concern metrics versus bug count for the Rhino project. DOSC and DOSM appear to have a logarithmic relationship with bug count. CDC, CDO, and LOCC have a clear linear relationship with bug count. We therefore used Spearman's rank-order correlation coefficient, which supports both linear and curvilinear relationships, and mitigates to a certain extent the unreliability of our measurements (we discuss this further in Section 7.1). Table 4 shows our correlation results for the three projects. Correlation coefficients range from -1.00 (a perfect negative correlation) to $+1.00$ (a perfect positive correlation). A coefficient of 0 means no correlation.

The Mylyn-Bugzilla results (see Table 4a) show that our DOS metrics (DOSC and DOSM) are moderately correlated with bug count (0.39 and 0.50) and the concern diffusion metrics (CDC and CDO) are strongly[15] correlated (0.57 and 0.61). These correlations were statistically significant at the 5 percent confidence level. In other words, there is a small (5 percent) probability that the relationship between the

15. Our use of the qualitative descriptions of correlation strength, "strong" and "moderate," is based on convention [14, pp. 79-80].

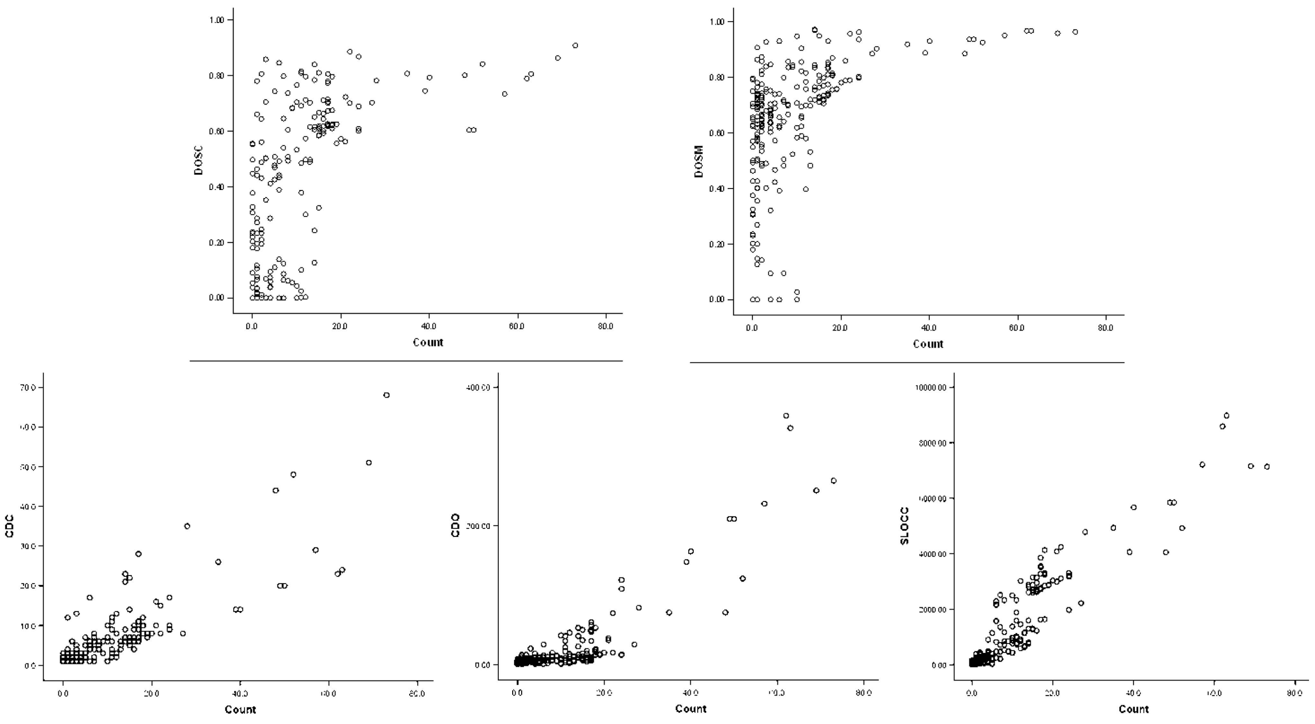


Fig 7. Scatter plots of the concern metrics versus bug count for Rhino.

scattering metrics and the bug count is coincidental. Thus, even though we obtained only 28 concerns for Mylyn-Bugzilla instead of 30, as prescribed in Section 4.1, our results are still statistically significant.

The correlations are stronger and more statistically significant for Rhino (see Table 4b). All scattering metrics (DOSC, DOSM, CDC, and CDO) have substantial correlation coefficients—ranging from 0.65 to 0.74—indicating a strong association with bug count. The probability that the association exists by chance is minute (less than 0.01 percent).

For iBATIS (see Table 4c), we see correlations of similar strength as Mylyn-Bugzilla. All scattering metrics have a nontrivial association with defects with correlation coefficients ranging from 0.29 for DOSM to 0.58 for CDC. Similarly to Rhino, the probability that the association exists by chance is minute (less than 0.01 percent).

Taken together, these results support our hypothesis:

Concern scattering is correlated with defects.

It is interesting to consider one of the crosscutting concerns revealed by our analysis. In Rhino, the requirement "10.1.4—Scope Chain and Identifier Resolution" was the most scattered according to its DOSC (0.91) and CDC (68) values. This requirement specifies the scoping rules for identifier lookup in ECMAScript. Its physical realization in the source code entails passing around a scope parameter to most of the method calls in Rhino, resulting in the concern being scattered across the code base. Considering its highly scattered nature, it is not surprising that the concern is also the most error prone (73 bugs).

15. Our use of the qualitative descriptions of correlation strength, "strong" and "moderate," is based on convention [14, pp. 79-80].

6.2 Correlations between the Scattering Metrics

From Tables 4a, 4b, and 4c, we observe that the scattering metrics are strongly correlated with each other. For example, for Rhino, CDC is almost perfectly correlated with DOSC (0.96). This is expected since CDC and CDO are coarser versions of 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.

Although we were hoping to determine if it is more profitable to analyze scattering at the class or method level when correlating defects, our results were inconclusive. For Mylyn-Bugzilla and Rhino, method-level scattering (CDO) had the strongest correlation (0.61 and 0.77, respectively), whereas, for iBATIS, class-level scattering (CDC) was the strongest (0.58).

By and large, CDC and CDO were more strongly correlated with defects than DOSC and DOSM. 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. However, our results indicate that simply knowing the *number* of classes and methods involved in the implementation of a concern is sufficient. It may be that degree of scattering is more useful when concern assignment is performed at the level of statements (or below). For example, moving redundant code into a shared function reduces degree of scattering, but is undetected by CDC and CDO.

6.3 Testing for the Confounding Effect of Size

For all the projects, the size of the concern implementation (LOCC) had the strongest or second strongest correlation

TABLE 4
Spearman Correlation Coefficients (a) Mylyn-Bugzilla, (b) Rhino, (c) iBATIS

	DOSM	*CDC*	*CDO*	*LOCC*	**Bugs**
DOSC	.64	.84	.57	.38	**.39**
DOSM	—	.77	.91	.63	**.50**
CDC	—	—	.78	.65	**.57**
CDO	—	—	—	.71	**.61**
LOCC	—	—	—	—	**.77**

(a)

	DOSM	*CDC*	*CDO*	*LOCC*	**Bugs**
DOSC	.62	.96	.74	.60	**.67**
DOSM	—	.63	.88	.68	**.66**
CDC	—	—	.80	.67	**.73**
CDO	—	—	—	.80	**.77**
LOCC	—	—	—	—	**.90**

(b)

	DOSM	*CDC*	*CDO*	*LOCC*	**Bugs**
DOSC	.67	.90	.73	.43	**.46**
DOSM	—	.67	.90	.64	**.29**
CDC	—	—	.78	.55	**.58**
CDO	—	—	—	.77	**.44**
LOCC	—	—	—	—	**.53**

(c)

In (a) all values are statistically significant at the 5 percent level (two-tailed). The sample size N (number of concerns) is 28. In (b), all values are statistically significant at the 0.01 percent level (two-tailed). The sample size N (number of concerns) is 357. In (c), all values are statistically significant at the 0.01 percent level (two-tailed). The sample size N (number of concerns) is 132.

with bug count (0.77, 0.90, and 0.53). This is consistent with several other studies [8], [11], [33] that found strong correlations between size metrics and defects (although Fenton and Ohlsson [23] found no correlation). This indicates that larger concerns have more defects. This also suggests that a refactoring that reduces scattering but increases concern size might actually increase defects.

Previous studies have found correlations between object-oriented metrics, such as the CK metrics [13] and fault-proneness. However, El Emam et al. showed that after controlling for the confounding effect of size the correlation disappeared [22]. The reason is that many object-oriented metrics are strongly correlated with size and therefore serve as surrogates for size.

Looking at Tables 4a, 4b, and 4c we see a strong correlation between the scattering metrics and size (LOCC). For example, for Rhino (Table 4b), the strength of the correlation between CDO and LOCC is very strong (0.80). The reason is obvious if one considers that, as more classes and methods become involved in a concern's implementation (CDC and CDO), the number of lines (LOCC) grows. In fact, CDC and CDO cannot increase without a simultaneous increase in LOCC. DOSC and DOSM 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. There is, however, a significant correlation between these metrics and size, ranging from 0.38 to 0.68.

The strong correlation between all of the scattering metrics and size and between size and bug count indicates that we must test for a confounding effect. For the sake of thoroughness, we performed two tests: *stepwise regression analysis* and *principal component analysis (PCA)*.

6.3.1 Size Test 1: Stepwise Regression Analysis

For stepwise regression analysis [37, pp. 263-264], we build a regression model that initially consists of the concern metric that has the single largest correlation with bug count. We then add metrics to the model based on their partial correlation with the metrics already in the model. With each new set of metrics, the model is evaluated and metrics that do not significantly contribute toward the statistical significance are removed so that, in the end, the best set of metrics that explain the maximum possible variance is left. The amount of variance explained by a model is signified by the model's R^2 value [37, p. 229].

For completeness, we also include the Adjusted R^2 and Standard Error of Estimate values. Adjusted R^2 explains any bias in the R^2 measure by taking into account the degrees of freedom of the predictor variables and the sample population. From Tables 5 and 6, we see that the Adjusted R^2 values are almost the same as the R^2 values, which indicates that the bias is absent from our models. Standard Error of Estimate (Std. Error) measures the deviation of the actual bug count from the bug count predicted by the model.

We can now state our test: If size explains *all* of the variance in bug count, we would not expect stepwise regression to include any of our scattering metrics. Table 5 shows our stepwise regression results for the three projects. Narrowing our focus to Mylyn-Bugzilla (Table 5a), we see that the stepwise regression completed after two steps. From the first step, we can see that the most significant metric is size (LOCC). The R^2 value of 0.73 means that we can explain 73.0 percent of the variance in bug count using size alone. The second step adds a scattering metric (CDC), which improves R^2 only slightly.

TABLE 5
Stepwise Regression Model Summaries: (a) Mylyn-Bugzilla, (b) Rhino, (c) iBATIS

(a)

Model	R^2	Adjusted R^2	Std. Error
1[a]	.73	.72	3.89
2[b]	.79	.77	3.49

(b)

Model	R^2	Adjusted R^2	Std. Error
1[a]	.92	.92	11.42
2[b]	.92	.92	11.28
3[c]	.93	.93	10.96

(c)

Model	R^2	Adjusted R^2	Std. Error
1[a]	.80	.80	1.75
2[b]	.82	.82	1.67
3[c]	.83	.83	1.62
4[d]	.84	.84	1.59
5[e]	.85	.85	1.54

For (a): [a] Metrics Used: LOCC, [b] Metrics Used: LOCC, CDC.
For (b): [a] Merics Used: LogDOSC, [b] Metrics Used: LogDOSC, CDC, [c] Meetrics Used: LogDosc, CDC, LogDOSM.
For (c): [a] Metrics Used: SLOCC, [b] Metrics Used: SLOCC, CDO, [c] Metrics Used: SLOCC, CDO, CDC, [d] Metrics Used: SLOCC, CDO, CDC, DOSM, [e] Metrics Used: SLOCC, CDO, CDC, DOSM, DOSC.

TABLE 6
Principal Components: (a) Mylyn-Bugzilla, (b) Rhino, (c) iBATIS

(a)

	Component		
	1	2	3
DOSC	**.70**	.68	-.16
DOSM	**.80**	.22	.55
CDC	**.92**	.11	-.32
CDO	**.92**	-.32	-.03
LOCC	**.80**	-.58	-.02

(b)

	Component			
	1	2	3	4
DOSC	**.81**	.26	-.50	-.06
DOSM	**.65**	.67	.35	.03
CDC	**.89**	-.21	-.05	.39
CDO	**.83**	-.40	.30	-.10
SLOCC	**.92**	-.15	-.03	-.26

(c)

	Component		
	1	2	3
DOSC	**.55**	.68	-.48
DOSM	**.49**	.73	.48
CDC	**.96**	-.18	-.07
CDO	**.94**	-.30	.03
SLOCC	**.94**	-.30	.07

Shifting our attention to iBATIS (Table 5c), we find results similar to Mylyn-Bugzilla. While all of the scattering metrics in the final model (Model 5) contribute to explaining the variance in bug count to some extent, size explains the most variance.

This is to be expected. Our theory states that scattering is responsible for *some*—not *all*—of the defects in the program. We therefore expect that size explains most of the defects. The stepwise regression models for Mylyn-Bugzilla and iBATIS indicate that scattering explains *some* of the variance in bug count, which supports our hypothesis.

The stepwise regression results for Rhino (Table 5b) are strongly in favor of our hypothesis. As explained in Section 6.1, DOSC and DOSM have a clear logarithmic relationship with bug count for Rhino. Since stepwise regression expects a linear relationship, we first took the logarithm of these metrics, which explains the terms LogDOSC and LogDOSM in Table 5b. From the table, we see that the regression terminated after three steps and that, at each step, the metric that explains most of the remaining variance was chosen. From the R^2 value, we see that the model with scattering metrics LogDOSC, CDC, and LogDOSM explains 92.8 percent of the variance in bug count—*size (LOCC) does not factor into the prediction at all.*

In summary, stepwise regression analysis supports our hypothesis because it indicates that scattering explains *some*—and, for Rhino, *most*—of the variance in bug count for the three projects we studied *independent of size*.

6.3.2 Size Test 2: Principal Component Analysis

Because the scattering metrics and LOCC are highly correlated among themselves, it is likely that the Spearman and stepwise regression models do not explain as much of the variance in bug count as the coefficients imply (i.e., they overfit the data). To overcome this *collinearity*, we used principal component analysis (PCA) [36]. With PCA, a small number of uncorrelated weighted combinations of metrics (that account for as much sample variance as possible) are generated such that the transformed variables are independent. These weighted combinations of metrics are called *principal components*.

Running PCA on the metrics for the three projects resulted in the generation of the principal components shown in Tables 6a, 6b, and 6c, which account for greater than 95 percent of the sample variance. For Mylyn-Bugzilla (Table 6a), three components were generated. The first component explains the highest amount of variance, the second component explains the second highest, and so on. The first component weighs all of the metrics highly—DOSC has a weighting of 0.70, DOSM has 0.80, and so forth. This indicates that the scattering metrics are significant contributors to explaining the variance in bug count. The results for Rhino (Table 6b) and iBATIS (Table 6c) are similar.

We then used the principal components to build a regression model for each project. From Table 7, we see that the models are highly accurate at predicting bug count—as indicated by the high R^2 values—further indicating the

TABLE 7
PCA Regression Model Summary

	R^2	Adjusted R^2	Std. Error
Mylyn–Bugzilla	0.78	0.75	3.69
Rhino	0.92	0.92	3.03
iBATIS	0.78	0.78	1.86

importance of the scattering metrics. The models are also statistically significant at the 99 percent confidence level.

From the stepwise regression analysis and PCA results, we conclude that concern size is not the single dominating factor—the scattering metrics contribute toward explaining the variance in bug count, thus signifying their importance and reaffirming our hypothesis.

6.4 Do Crosscutting Concerns Cause Defects?

A correlation by itself does not imply causality [37, p. 213]. Isolating cause and effect is easier for controlled experiments than for correlation studies such as our own [38, pp. 80-81]. Kan outlines three *criteria for causality* that correlation studies must meet before making causality claims [38, pp. 80-81]. The first is that *the cause must precede the effect.* This is equivalent to saying that crosscutting concerns precede defects related to those concerns. In our theory, crosscutting first manifests itself in the *concern implementation plan*, which precedes the defects that are introduced during the implementation of that plan.

The second criterion is that *a correlation must exist.* The results of our three case studies indicate that a moderate to strong statistically significant correlation exists between scattering and defects.

Finally, *the correlation must not be spurious.* We argue that the correlation is not spurious because 1) there is a plausible reason (i.e., a theoretical justification) for the correlation to exist, 2) we verified that the scattering metrics are not surrogates for size, and 3) the correlation is not coincidental since we observed similar correlation results for three separate case studies.

This brings us back to our original question: Do crosscutting concerns cause defects? Our theory and the results of our studies suggest "yes." Independent verification in the form of empirical studies and controlled experiments is needed before we can be confident that a causal relationship exists.

7 Threats to Internal Validity

7.1 Concern Assignment Unreliability

Our concern metrics are unreliable because of the subjectivity inherent in our concern assignment methodology. This limits the consistency and repeatability of our measurements. Indeed, studies [45], [52] have shown disparities between concern assignments produced by different analysts. Unreliability can also reduce the strength and significance of the relationship between scattering and defects [21].

While automated assignment techniques [2] produce consistent results, we believe that the assignment produced by our interactive technique more accurately captures the rationale behind the code [17], which we need before we can apply our theory. Thus, we tolerate some loss in measurement reliability for improved relevance.

We compensated for this unreliability in two ways. First, we used a rank-order correlation (Spearman) that can tolerate unreliable measurements as long as the relative ordering (rank) of the measurements is correct [38, p. 78]. Comparing measurements by relative order instead of absolute value is consistent with how the concentration metric upon which our DOS metric is based should be interpreted [62]. This implies that it is sufficient for the concern assignment to be a close approximation of the "correct" concern assignment.

Second, two of our studies had large sample sizes ($N = 357$ and 132). The correlation results show a moderate to strong statistically significant relationship between scattering and defects for all three case studies, which we would not expect if the measurements were completely unreliable.

Our future work is to measure the *reproducibility* (variance across analysts), *repeatability* (variance across trials), and *accuracy* (variance with respect to a reference assignment, i.e., a *gold standard*) of our prune dependency assignment technique. We will then be able to properly compensate for measurement errors by incorporating error estimates into our regression model.

7.2 Bug Assignment Errors

In the context of bug assignment, a *false positive* means that a bug *should not* have been associated with a program element and a *false negative* means a bug *should* have been associated with a program element (but was not). These false observations may perturb bug counts. Bug assignment errors fall into three categories:

1. Incorrect bug metadata.
2. Bugs mapped to the wrong elements.
3. Bugs mapped to "missing" elements.

7.2.1 *Incorrect Bug Metadata*

Many of the Mylyn-Bugzilla bugs were clearly enhancements although they were not classified as such. This is an example of a Category 1 error.

7.2.2 *Bugs Mapped to the Wrong Elements*

Category 2 errors can occur when a commit message is misleading. For example, a sequence of numbers may be mistaken for a valid bug ID (Category 2a, false positive) or a bug ID may be referenced coincidentally (Category 2b, false positive) or not at all (Category 2c, false negative). We eliminated Category 2a errors by validating all bug IDs against the ITS, which allowed us to eliminate 32 false positives for one project. By choosing projects that use bug IDs in commit messages in a disciplined and consistent (and, in the case of Mylyn-Bugzilla, completely automated) way, we believe there are no instances of Category 2b or 2c errors.

It is also possible that the real defect does not lie in the lines changed by the bug fix (Category 2d). For example, instead of fixing the defect (e.g., because it lies in a third-party library), a "workaround" is made to another part of the code so that the defect no longer manifests itself. This leads to a false positive *and* negative since the bug should be mapped to a completely different program element. We agree with Purushothaman and Perry, who concluded that detecting Category 2d errors would require more information than *"is available or automatically inferable"* [51].

It is common for a bug fix to include modifications unrelated to the bug (Category 2e) [10], [21], [51] or fixes for multiple bugs (Category 2f). To reduce Category 2e errors, we ignored insignificant changes such as changes to whitespace and comments. We also ignored bug fixes associated with the first version of a file. To avoid false positives due to Category 2f errors, we ignored bug fixes that referenced multiple bug IDs.

For two case studies, the bug assignment task was completely automated using BugTagger. This eliminates Category 2 errors caused by the inconsistencies inherent in a manual assignment and guarantees assignment repeatability. However, there is always the possibility that BugTagger is faulty. We used the Jaccard similarity metric [58] to compare the bug assignment produced by BugTagger with the one we created by hand for Mylyn-Bugzilla and found the Jaccard similarity was 0.87, indicating the assignments were very similar. On closer inspection, we found that many of the disagreements were due to human errors made during manual assignment, further vindicating our decision to mechanize.

7.2.3 Bugs Mapped to "Missing" Elements

Category 3 errors can occur when a bug is mapped to some methods and fields that were present at the time of the bug fix but were subsequently removed or renamed. For Rhino, initially 37 bugs (21 percent) were mapped *entirely* to missing methods and fields and therefore could not be associated with any concern. Our assignment technique uniquely identifies program elements by their *signature* (the fully qualified element name and, in the case of methods, the list of parameters). Therefore, we investigated the possibility that the element's signature had been changed, e.g., the element was renamed or the parameter list was modified. BugTagger automatically detects changes to the parameter list, e.g., foo(int, int) changed to foo(float, float); however, name changes were harder to detect automatically. We therefore tracked down name changes by hand and were able to halve the number of bugs that could not be mapped to any concern. To further reduce these errors, we would need a concern mapping for every revision—not just the latest.

We would like to point out that the studies we are aware of that analyze defects based on mining software repositories [16], [22], [27], [50], [57] suffer from the same problems. We believe that our enumeration of the possible errors and recommendations for avoiding them will be a welcome contribution to this area of research.

7.2.4 Impact of Assignment Errors on Our Results

Ultimately, we care about the extent to which false bug-*code* assignments propagate to false bug-*concern* assignments, which will increase errors in our defect counts and correlations. Some bug assignment errors may be masked. For example, we may miss a program element that should have been assigned to the bug, but, as long as another assigned program element causes the bug to be associated with the correct concern, the false negative is masked. False positives can be masked similarly.

A bug that is not associated with a concern is not necessarily a problem. For example, in Mylyn-Bugzilla, 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.

7.3 Assignment Aggregation Error

Our concern and bug assignment techniques aggregate at the member level the LOCs associated with the concern or defect. This loss in granularity makes our assignments less precise [65]. For example, often *sibling leaf concerns*[16] in Rhino are implemented using switch statements.[17] For instance, the parent concern "15.2.4—Properties of Object Prototypes" has the following child concerns:

- 15.2.4.1—constructor;
- 15.2.4.2—toString;
- 15.2.4.3—toLocaleString;
- 15.2.4.4—valueOf.

In this case, even though each concern is really only associated with an individual case in the switch statement [19], they will be assigned at the method level and will therefore have inflated concern sizes. Let us further suppose a bug is associated with one of the cases. The bug will also be assigned at the method level and will therefore inflate the bug counts for the concerns not associated with the case statement.

In addition to inflated sizes and bug counts, the metrics computed for the concern subset will be very similar. For the Rhino project, we found that the standard deviations for all of the metrics were much lower for sibling leaf concern clusters than for the entire population of concerns. For example, the standard deviation of the bug count was 3.31 for sibling leaf concerns but 14.07 for the entire population.

It is hard to predict the impact that aggregation error has on our results. Aggregation error appears to be biased in favor of supporting our main hypothesis in the sense that the more scattered a concern is, the more methods contribute to its implementation, increasing the number of opportunities for aggregation error to inflate the concern size and defect count.

Despite this bias, we argue that eliminating aggregation error would not reverse our conclusions for the following reason: It only occurs when method level assignment is not granular enough to faithfully represent the implementation of a concern. This is true for Rhino, where concerns were

16. Sibling leaf concerns have the same parent and no children.
17. Rhino inherits many of these quirks from the JavaScript interpreter it was based on, which was written in C. If Rhino had been written in Java from scratch, virtual methods might have been used instead of switch statements, which would have reduced aggregation error.

very fine grained and switch statements were prevalent. However, this is not the case for Mylyn-Bugzilla and iBATIS. Since the correlations for all three studies were moderate to strong and statistically significant, we conclude that eliminating aggregation error would not reverse our conclusions.

8 Threats to External Validity

External validity is the degree to which we can draw general conclusions from our results. 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 [4]. El Emam concurs: *"It is only when evidence is accumulated that a particular metric is valid across systems and across organizations can we draw general conclusions"* [21].

There are many possible sources of defects, including the complexity of the problem domain, developer experience, mental and physical stress, tool support, etc. [21]. Our theory only attempts to explain a small portion of defects, namely, those caused by the complexity associated with implementing crosscutting concerns.

We expect that the programming language has a large impact on how scattered a concern's implementation is. Because the programs we studied were written in Java, we cannot generalize our results to other programming languages. Interesting future work would be to compare Rhino with SpiderMonkey,[18] an implementation of ECMAScript written in C.

The open source projects we studied had many similarities to projects developed in industry including the use of change management systems, extensive test suites, and descriptive commit messages. Therefore, we expect our results to hold in an industrial setting for Java programs of similar size (13-44 KSLOCs).

It is possible that the relationship between scattering and defects only holds for requirement concerns and not for concerns from other domains, such as the ones mentioned in Section 2.

9 Literature Review

9.1 Feature Location

The goal of feature location (or, more generally, *concern location*) is to learn the *how*, *where*, and *why* of software: How and where is a feature implemented? Why is the code implemented this way? In a study of the information needs of developers, Ko et al. concluded that the information most sought after—and most difficult to obtain—was *"the intent behind existing code and code yet to be written"* [44]. This information is essential for making changes correctly, yet is largely undocumented.

Researchers have employed a variety of automated techniques to recover links between concerns and code. Antoniol et al. employed *information retrieval* (IR) to find the correspondence between requirement documents and identifiers and comments in the source code [2]. Zhao et al. augment IR results with branch-reserving call graph information to improve relevancy [65]. Several researchers [20], [61], [62] have analyzed *execution traces* of the program to see which methods are called when a feature is invoked. Poshyanyk et al. showed that accuracy can be improved by combining static and dynamic analysis techniques [50].

Automation is essential for feature location to scale to large continuously evolving systems. However, the relevancy of the concern-code mapping they produce is debatable. For example, links between concerns and code may be missed by IR techniques if meaningful identifier names are not used [50], [65] and by execution tracing techniques if features cannot be exercised completely and orthogonally [50].

While the mappings produced by these automated techniques are well suited for guiding program comprehension and maintenance activities, we felt they would not be sufficiently relevant for validating our theory. Furthermore, we sought to eliminate the possibility that deficiencies, mistakes, or biases in the assignment algorithm could skew our results. We therefore required all assignment decisions to be made by a human analyst using our interactive concern assignment tool, ConcernTagger, as explained in Section 4.2. Obviously, this limited the size of the programs we could analyze.

9.2 Empirical Studies of Crosscutting Concerns

Many 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.

For example, some researchers [45], [52], [64] have created concern metrics that measure scattering in absolute terms (e.g., number of classes that contribute to the implementation of the concern). Garcia et al. used their *concern diffusion metrics* in several studies [25], [28], [30] to show that, in general, modularizing crosscutting concerns using aspect-oriented programming improves the separation of concerns. As explained in Section 3.3, we believe our *DOS metrics* complement the concern diffusion metrics by providing a more fine-grained measurement of scattering.

We know of one study besides our own that correlates aspect and 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 [3]. 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.

18. http://www.mozilla.org/js/spidermonkey.

9.3 Empirical Studies of Software Quality

In this section, we discuss some of the earlier work related to investigations of using historical measures of complexity, code churn, prerelease defects, etc., as predictors of software quality in large software systems.

Letovsky and Soloway [46] use file status information such as "new," "changed," and "unchanged" along with other explanatory variables such as LOCs, age, prior faults, etc., as predictors in a negative binomial regression equation to predict the number of faults in a multiple-release software system. The predictions made using their binomial regression model had high accuracy for faults found in both early and later stages of development [46]. Khoshgoftaar et al. [40] studied two consecutive releases of a large legacy system for telecommunications. The system contained over 38,000 procedures in 171 modules. Discriminant analysis identified fault-prone modules based on 16 static software product metrics with types 1 and 2 misclassification rates of 21.7 percent and 19.1 percent, respectively, and an overall misclassification rate of 21.0 percent. Nagappan and Ball [49] investigated the use of a set of relative code churn measures in isolation as predictors of software defect density for the Windows Server 2003 system. Relative churn measures are normalized values of the various measures obtained during the measurement of churn. They found that relative code churn measures were strong statistically significant predictors of code quality. In contrast, Eisenbarth et al. found no correlation between code churn and code quality [19].

Several researchers have attempted to find a relationship between defects and internal product metrics, such as code churn [49], size metrics [11], [22], [23], [33], object-oriented metrics (e.g., the CK metrics [13]) [11], [22], [33], design metrics [11], and prerelease defects [7]. We add to this body of research by examining the relationship between concern metrics and defects.

10 Conclusion

This paper is the first to provide empirical evidence suggesting that crosscutting concerns cause defects. We examined the concerns of three small to medium-sized open-source Java projects and found that the more scattered the implementation of a concern is, the more likely it is to have defects. Moreover, this effect is evident independent of the size of the concern's implementation (in terms of LOCs).

This evidence, although preliminary, is important for several reasons. It adds credibility to the claims about the dangers of crosscutting concerns made by the aspect-oriented programming and programming language communities. By establishing a correlation between concern metrics and an external quality indicator—*defects*—we provide a stronger form of validation for these metrics than previous empirical studies that focused on internal quality indicators (e.g., [25], [28]).

We also proposed a theory that suggests why crosscutting concerns might cause defects and described our concern model and metrics. These can serve as the foundation for future empirical work.

It is important to realize that the novelty of our experiment and the subjectivity inherent in our methodology limit the conclusions we can draw from our results. Further studies are needed before we can draw general conclusions about the relationship between scattering and defects. To facilitate this, we are working on automating our concern assignment technique, which is needed to make application of our metrics practical for large systems (greater than 50 KLOCs).

Several questions remain. Can we reduce the likelihood of defects by reducing crosscutting (assuming concern size does not increase)? Are crosscutting concerns a byproduct of programming technology, developer aptitude, or the inherent complexity of the concern? What is the relationship between code churn and scattering? If a relationship exists, we can use code churn to help identify crosscutting concerns [10] 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 (e.g., by modularizing the underlying crosscutting concerns).

Appendix

Online Appendix

We invite researchers to replicate our case studies. Source code for the subject programs and our measurement tools, complete concern and bug lists, concern-code and bug-code mappings, and our results are available at http://www.cs.columbia.edu/~eaddy/concerntagger.

Acknowledgments

This research was funded in part by the Natural Sciences and Engineering Research Council of Canada. The authors are extremely grateful to the anonymous reviewers for their feedback and suggestions.

References

[1] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools,* second ed. Addison Wesley, 2006.
[2] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering Traceability Links between Code and Documentation," *IEEE Trans. Software Eng.,* vol. 28, no. 10, pp. 970-983, Oct. 2002.
[3] M. Bartsch and R. Harrison, "Towards an Empirical Validation of Aspect-Oriented Coupling Measures," *Proc. Workshop Assessment of Aspect Techniques,* 2007.
[4] V. Basili, F. Shull, and F. Lanubile, "Building Knowledge through Families of Experiments," *IEEE Trans. Software Eng.,* vol. 25, no. 4, pp. 456-473, July/Aug. 1999.
[5] K.v.D. Berg, J.M. Conejero, and J. Hernández, "Analysis of Crosscutting across Software Development Phases Based on Traceability," *Proc. Workshop Aspect-Oriented Requirements Eng. and Architecture Design (Early Aspects),* 2006.
[6] T.J. Biggerstaff, B.G. Mitbander, and D. Webster, "The Concept Assignment Problem in Program Understanding," *Proc. 15th Int'l Conf. Software Eng.,* pp. 482-498, 1993.
[7] S. Biyani and P. Santhanam, "Exploring Defect Data from Development and Customer Usage on Software Modules over Multiple Releases," *Proc. Ninth Int'l Symp. Software Reliability Eng.,* 1998.
[8] L. Briand, J. Wuest, J. Daly, and V. Porter, "Exploring the Relationships between Design Measures and Software Quality in Object Oriented Systems," *J. Systems and Software,* vol. 51, pp. 245-273, 2000.

[9] M. Bruntink, A.v. Deursen, and T. Tourwé, "Discovering Faults in Idiom-Based Exception Handling," *Proc. 28th Int'l Conf. Software Eng.,* 2006.

[10] G. Canfora, L. Cerulo, and M.D. Penta, "On the Use of Line Co-Change for Identifying Crosscutting Concern Code," *Proc. 22nd Int'l Conf. Software Maintenance,* 2006.

[11] M. Cartwright and M. Shepperd, "An Empirical Investigation of an Object-Oriented Software System," *IEEE Trans. Software Eng.,* vol. 26, no. 8, pp. 786-796, Aug. 2000.

[12] P. Cederqvist et al., *Version Management with CVS.* Network Theory, Ltd., 2002.

[13] 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.

[14] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences,* second ed. Lawrence Erlbaum Assoc., 1988.

[15] B. Collins-Sussman, B.W. Fitzpatrick, and C.M. Pilato, *Version Control with Subversion.* O'Reilly, 2004.

[16] D. Čubranić, G.C. Murphy, J. Singer, and K.S. Booth, "Hipikat: A Project Memory for Software Development," *IEEE Trans. Software Eng.,* vol. 31, no. 6, pp. 446-465, June 2005.

[17] M. Eaddy, A. Aho, and G.C. Murphy, "Identifying, Assigning, and Quantifying Crosscutting Concerns," *Proc. Workshop Assessment of Contemporary Modularization Techniques,* 2007.

[18] ECMA, "ECMAScript Standard," vol. ECMA-262 v3, ISO/IEC 16262, 2007.

[19] T. Eisenbarth, R. Koschke, and D. Simon, "Locating Features in Source Code," *IEEE Trans. Software Eng.,* vol. 29, no. 3, pp. 210-224, Mar. 2003.

[20] A.D. Eisenberg and K. De Volder, "Dynamic Feature Traces: Finding Features in Unfamiliar Code," *Proc. 21st Int'l Conf. Software Maintenance,* pp. 337-346, 2005.

[21] K. El Emam, "A Methodology for Validating Software Product Metrics," Technical Report NRC 44142, Nat'l Research Council of Canada, 2000.

[22] K. El Emam, S. Benlarbi, N. Goel, and S.N. Rai, "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics," *IEEE Trans. Software Eng.,* vol. 27, no. 7, pp. 630-650, July 2001.

[23] N.E. Fenton and N. Ohlsson, "Quantitative Analysis of Faults and Failures in Complex Software Systems," *IEEE Trans. Software Eng.,* vol. 26, no. 8, pp. 797-814, Aug. 2000.

[24] E. Figueiredo, A. Garcia, C. Sant'Anna, U. Kulesza, and C. Lucena, "Assessing Aspect-Oriented Artifacts: Towards a Tool-Supported Quantitative Method," *Proc. Ninth ECOOP Workshop Quantitative Approaches in Object-Oriented Software Eng.,* 2005.

[25] 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," *Foundations of Software Eng.,* pp. 152-162, 2006.

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

[27] M. Fischer, M. Pinzger, and H. Gall, "Populating a Release History Database from Version Control and Bug Tracking Systems," *Proc. 19th Int'l Conf. Software Maintenance,* pp. 23-32, 2003.

[28] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A.v. Staa, "Modularizing Design Patterns with Aspects: A Quantitative Study," *Proc. Int'l Conf. Aspect-Oriented Software Development,* 2005.

[29] C. Gibbs, C.R. Liu, and Y. Coady, "Sustainable System Infrastructure and Big Bang Evolution: Can Aspects Keep Pace," *Proc. 19th European Conf. Object-Oriented Programming,* pp. 241-261, 2005.

[30] P. Greenwood, T.T. Bartolomei, E. Figueiredo, M. Dósea, A. Garcia, N. Cacho, C. Sant'Anna, S. Soares, P. Borba, U. Kulesza, and A. Rashid, "On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study," *Proc. 21st European Conf. Object-Oriented Programming,* pp. 176-200, 2007.

[31] J.E. Hannay, D.I.K. Sjøberg, and T. Dybå, "A Systematic Review of Theory Use in Software Engineering Experiments," *IEEE Trans. Software Eng.,* vol. 33, no. 2, pp. 87-107, Feb. 2007.

[32] R. Harrison, S. Counsel, and R. Nithi, "Experimental Assessment of the Effect of Inheritance on the Maintainability of Object-Oriented Systems," *J. Systems and Software,* vol. 52, pp. 173-179, 2000.

[33] R. Harrison, L. Samaraweera, M. Dobie, and P. Lewis, "An Evaluation of Code Metrics for Object-Oriented Programs," *Information and Software Technology,* vol. 38, pp. 443-450, 1996.

[34] *IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology,* IEEE, 1990.

[35] ISO/IEC, "Information Technology—Software Product Evaluation," IDS 14598-1, 1996.

[36] E.J. Jackson, *A User's Guide to Principal Components.* John Wiley & Sons, 1991.

[37] S.K. Kachigan, *Statistical Analysis.* Radius Press, 1986.

[38] S.H. Kan, *Metrics and Models in Software Quality Engineering,* second ed. Addison-Wesley, 2003.

[39] M. Kersten and G.C. Murphy, "Using Task Context to Improve Programmer Productivity," *Foundations of Software Eng.,* 2006.

[40] T.M. Khoshgoftaar, E.B. Allen, N. Goel, A. Nandi, and J. McMullan, "Detection of Software Modules with High Debug Code Churn in a Very Large Legacy System," *Proc. Seventh Int'l Symp. Software Reliability Eng.,* pp. 364-371, 1996.

[41] T.M. Khoshgoftaar, E.B. Allen, W.D. Jones, and J.P. Hudepohl, "Classification-Tree Models of Software Quality over Multiple Releases," *IEEE Trans. Reliability,* vol. 49, no. 1, pp. 4-11, 2000.

[42] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C.V. Lopes, C. Maeda, and A. Mendhekar, "Aspect-Oriented Programming," *ACM Computing Surveys,* vol. 28, no. 4es, p. 154, 1996.

[43] B. Kitchenham, S.L. Pfleeger, and N. Fenton, "Towards a Framework for Software Measurement Validation," *IEEE Trans. Software Eng.,* vol. 21, no. 12, pp. 929-944, Dec. 1995.

[44] A.J. Ko, R. DeLine, and G. Venolia, "Information Needs in Collocated Software Development Teams," *Proc. 29th Int'l Conf. Software Eng.,* 2007.

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

[46] S. Letovsky and E. Soloway, "Delocalized Plans and Program Comprehension," *IEEE Software,* vol. 3, no. 3, pp. 41-49, 1986.

[47] M. Lippert and C.V. Lopes, "A Study on Exception Detection and Handling Using Aspect-Oriented Programming," *Proc. 22nd Int'l Conf. Software Eng.,* pp. 418-427, 2000.

[48] A.v. Mayrhauser, A.M. Vans, and A.E. Howe, "Program Understanding Behaviour during Enhancement of Large-Scale Software," *Software Maintenance: Research and Practice,* vol. 9, pp. 299-327, 1997.

[49] N. Nagappan and T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density," *Proc. 27th Int'l Conf. Software Eng.,* 2005.

[50] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval," *IEEE Trans. Software Eng.,* vol. 33, no. 6, pp. 420-432, June 2007.

[51] R. Purushothaman and D.E. Perry, "Toward Understanding the Rhetoric of Small Source Changes," *IEEE Trans. Software Eng.,* vol. 31, no. 6, pp. 511-526, June 2005.

[52] M. Revelle, T. Broadbent, and D. Coppit, "Understanding Concerns in Software: Insights Gained from Two Case Studies," *Proc. 13th IEEE Int'l Workshop Program Comprehension,* 2005.

[53] M.P. Robillard, "Representing Concerns in Source Code," PhD thesis, Computer Science Dept., Univ. of British Columbia, Nov. 2003.

[54] M.P. Robillard, W. Coelho, and G.C. Murphy, "How Effective Developers Investigate Source Code: An Exploratory Study," *IEEE Trans. Software Eng.,* vol. 30, no. 12, pp. 889-903, Dec. 2004.

[55] M.P. Robillard and F. Weigand-Warr, "ConcernMapper: Simple View-Based Separation of Scattered Concerns," *Proc. Workshop Eclipse Technology eXchange,* 2005.

[56] M.J. Rochkind, "The Source Code Control System," *IEEE Trans. Software Eng.,* vol. 1, no. 4, pp. 364-370, 1975.

[57] J. Śliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes," *Proc. Workshop Mining Software Repositories,* 2005.

[58] P.H. Sneath and R.R. Sokal, *Numerical Taxonomy.* W.H. Freeman, 1973.

[59] S.M. Sutton Jr. and I. Rouvellou, "Concern Modeling for Aspect-Oriented Software Development," *Aspect-Oriented Software Development,* pp. 479-505, Addison-Wesley, 2005.

[60] S.L. Tsang, S. Clarke, and E. Baniassad, "An Evaluation of Aspect-Oriented Programming for Java-Based Real-time Systems Development," *Proc. Seventh Int'l Symp. Object-Oriented Real-Time Distributed Computing,* 2004.

[61] N. Wilde and M.C. Scully, "Software Reconnaissance: Mapping Program Features to Code," *J. Software Maintenance and Evolution: Research and Practice,* vol. 7, no. 1, pp. 49-62, 1995.

[62] W.E. Wong, S.S. Gokhale, and J.R. Horgan, "Quantifying the Closeness between Program Components and Features," *J. Systems and Software,* vol. 54, no. 2, pp. 87-98, 2000.
[63] M.V. Zelkowitz and D.R. Wallace, "Experimental Models for Validating Technology," *Computer,* vol. 31, no. 5, pp. 23-31, May 1998.
[64] C. Zhang and H.-A. Jacobsen, "Quantifying Aspects in Middleware Platforms," *Proc. Int'l Conf. Aspect-Oriented Software Development,* pp. 130-139, 2003.
[65] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "SNIAFL: Towards a Static Noninteractive Approach to Feature Location," *ACM Trans. Software Eng. and Methodology,* vol. 15, no. 2, pp. 195-226, 2006.

Marc Eaddy received the dual BS degree in electrical engineering and computer science from Florida State University in 1995 and the MS degree in computer science from Columbia University in 2001. From 1995 to 2003, he worked at News Internet Services, where he helped develop the TV Guide Online Listings, and at Thomson Financial, where he built real-time stock market data applications. He is currently working toward the PhD degree in computer science at Columbia University under Alfred Aho. His research goal is to better understand—and solve—the crosscutting concern problem. He is a student member of the IEEE.

Thomas Zimmermann received the Diploma degree in computer science from the University of Passau in 2004. He is currently a PhD candidate at Saarland University, Saarbrücken, Germany, and an assistant professor in the Department of Computer Science at the University of Calgary. In 2006, he was a summer intern at Microsoft Research, where he analyzed the bug database of Windows Server 2003. His research interests are in software evolution, mining software repositories, empirical software engineering, program analysis, and development tools. He is a student member of the IEEE and the IEEE Computer Society.

Kaitlin D. Sherwood received the BS degree in metallurgical engineering and the MS degree in general engineering from the University of Illinois at Urbana-Champaign in 1984 and 1996, respectively. She is currently working toward the MS degree in computer science at the University of British Columbia under Gail Murphy. She has extensive experience in high-tech industries. She is keenly interested in personal productivity. She has written two books on managing e-mail overload and is currently researching individual programmer productivity.

Vibhav Garg received the BS degree in electronics engineering from Bangalore University, India, the MS degree in information technology from Bond University, Gold Coast, Australia, and the MS degree in computer science from Columbia University, New York. He is currently a senior consultant at CGI Technologies. He is keenly interested in languages, compilers, and software engineering issues.

Gail C. Murphy received the BSc degree in computing science from the University of Alberta in 1987 and the MS and PhD degrees in computer science and engineering from the University of Washington in 1994 and 1996, respectively. From 1987 to 1992, she was a software designer in industry. She is currently an associate professor in the Department of Computer Science at the University of British Columbia. Her research interests are in software evolution, software design, and source code analysis. She is a member of the IEEE Computer Society.

Nachiappan Nagappan received the BTech degree from the University of Madras in 2001 and the MS and PhD degrees from North Carolina State University in 2002 and 2005, respectively. He is a researcher in the Software Reliability Research Group at Microsoft Research. His research interests include software reliability and measurement, statistical modeling, and defect analysis. He is a member of the IEEE and the ACM.

Alfred V. Aho received the PhD degree in electrical engineering and computer science from Princeton University. He is the Lawrence Gussman Professor of Computer Science in the Department of Computer Science at Columbia University, New York. His research interests include software engineering, programming languages, compilers, and algorithms. He is a fellow of the American Association for the Advancement of Science, the ACM, and the IEEE.