# The Effect of Testability on Fault Proneness

A case study of the Apache HTTP Server

Mark Hays, Jane Hayes
Department of Computer Science
University of Kentucky
Lexington, USA
*mahays0@engr.uky.edu, hayes@cs.uky.edu*

*Abstract*—**Numerous studies have identified measures that relate to the fault-proneness of software components. An issue practitioners face in implementing these measures is that the measures tend to provide predictions at a very high level, for instance the per-module level, so it is difficult to provide specific recommendations based on those predictions. We examine a more specific measure, called software testability, based on work in test case generation. We discuss how it could be used to make more specific code improvement recommendations at the line-of-code level. In our experiment, we compare the testability of fault prone lines with unchanged lines. We apply the experiment to Apache HTTP Server and find that developers more readily identify faults in highly testable code. We then compare testability as a fault proneness predictor to McCabe's cyclomatic complexity and find testability has higher recall.**

***Keywords-fault proneness; testing; code coverage; static analysis***

## I. Introduction

The importance of fault proneness metrics is usually explained with succinct reasons, such as "methodologies and techniques for predicting the testing effort, monitoring process costs, and measuring results can help in increasing efficacy of software testing" [1]. These reasons make assumptions about the importance of software testing that outside observers may not share. We hear anecdotal stories from local software development firms where clients state during negotiations that they refuse to pay for time spent on testing, including the types of tests that test-driven development involves (such as unit tests).

During negotiations, their clients cite two issues with paying the developers to test. First, they see no reason why highly paid and qualified developers should make errors. Second, they believe that testing performed by the developers presents a conflict of interest. A series of Dilbert comics parodies this concern, where the developers are told they will earn $10 for every bug fixed [2]. From the client's perspective, the developers are no different from Dilbert. These two issues give clients pause and thus our local development companies simply see no reason to train developers to test, much less hire designated quality assurance staff.

Inspired by this problem, we issue a challenge to the fault proneness community to use their huge collection of metrics to propose actionable development plans to improve code quality. In the same vein as formal specification, rather than *testing* more, we propose *developing* more to reduce the testing burden of proof. To this end, we describe a novel fault proneness metric based on previous work in test case generation. Our contributions in this paper include: our position on using fault proneness metrics to improve code quality, our new testability metric, our tools, and our experiment.

In Section II, we discuss related work. In Section III, we describe our metric, *testability*, and illustrate our vision of how a developer could use certain fault proneness metrics to systematically reduce fault proneness. In Section IV, we describe a case study performed on the Apache HTTP Server and discuss our surprising results.

## II. Related Work

This section discusses related work in fault proneness, testabilty, and automated test case generation.

### *A. Fault proneness*

While numerous studies have been published on high-level fault proneness, Hata et al. claim to be the first to examine fault proneness on the per-line level. Their approach was vocabulary-based: they trained a machine learner on the text of fault-prone lines, then looked for files containing lines with similar vocabulary. They backtested their work by training the machine learner on Eclipse changesets. They used the changesets to predict which modules in Eclipse would contain faults. To assess their results, they measured recall and precision on the per-module level. They surveyed other work to compare their results [3].

On the module level, Hata et al. stated high recall and precision. Unfortunately, the results did not fit the granularity of the estimator. The estimator was trained to identify specific problem lines, but Hata et al. did not state how well their classifier identified the actual faulty lines. This oversight could easily be corrected in future work using a more advanced experiment design.

McCabe introduced a seminal fault-proneness measure called cyclomatic complexity. Given a control flow graph with one entry and one exit node that are not strongly connected to each other, McCabe defines cyclomatic complexity as:

$$M = E - N + 2 \tag{1}$$

where E is the number of edges and N is the number of nodes in the graph. To simplify testing, McCabe proves code transformations can reduce this complexity, but does not give guidance as to which transformation is optimal at a given point [4]. In our experiment, we compared McCabe's cyclomatic complexity to our per-line analysis of fault proneness, called testability.

## B. Testability

Voas first introduced the notion of software testability when examining the issue of measuring fault size. He identified three requirements of testable faults: execution (the code is executed), infection (the code performs a bad computation), and propagation (the output is corrupted by the bad computation). He proposed a randomized algorithm for dynamically inferring the testability of code. The weakness of this dynamic approach was that the tester had to write the universe of test cases for a given program to measure testability. In evaluating his work, Voas only examined the accuracy of his execution computation [5].

Freedman addressed the problem of the lack of propagation of variables in testing and created *domain testability* to measure it. Freedman formalized two desired qualities of domain-testable programs: observability (the program under test is purely a function of its parameters, it uses no global state) and controllability (the program can output all values in its defined range). Freedman measured domain testability by creating a domain-testable version of the program and then counting how many parameters he had to add; the more added parameters, the worse the domain testability. Freedman's students evaluated the measure by coding and testing programs according to specifications written with and without domain testability in mind. Freedman found that domain-testable specifications help speed up development [6].

## C. Automated test path construction

Li et al. examined algorithms for reducing coverage requirements into a smaller set of actual test paths. When graph-based coverage criteria produce many requirements, sometimes there is overlap in the requirements, causing the criteria to overestimate the amount of work required to test the program. Li et al. cast the problem of minimizing coverage requirements into a prefix graph. Li et al. detailed algorithms to extend the criteria into full test paths. They provided a tool, `Graph-Coverage`, for use in our research [7].

NASA's Java Path Finder uses formal methods to construct all feasible paths through a function. Unlike Li et al.'s tool, Path Finder filters out impossible paths and generates concrete test values to follow those paths. The downside is that Path Finder cannot abstract I/O operations, so it is limited to operations on numbers and strings [8]. If Path Finder were to overcome this limitation, we see use for it in the following definition of testability.

## III. Testability

We define testability with an intuitive static approximation to the Voas execution probability. For each function in the software, we generate the function's control flow graph. We feed the graph to the `Graph-Coverage` tool to compute the minimum set of paths required to achieve node coverage. We overlay all of the paths onto the graph. We define the testability of each node in the graph as the proportion of paths passing through that node. A testability of 1 means the node has perfect testability. A testability of 0 means the node is unreachable.

## A. Formal definition

Let $\texttt{Graph-Coverage}(G, \texttt{NODE})$ be the minimum set of test paths satisfying node coverage for graph *G*. Let $\texttt{Graph-Coverage}(G, \texttt{NODE})_b$ be the subset of $\texttt{Graph-Coverage}(G, \texttt{NODE})$ test paths containing node *b*. We define the testability of node *b* as:

$$t(b) = \frac{|\texttt{Graph-Coverage}(G, \texttt{NODE})_b|}{|\texttt{Graph-Coverage}(G, \texttt{NODE})|} \,. \tag{2}$$

`Graph-Coverage` refers to the Li et al. prefix graph algorithm mentioned in the related work. It can be substituted for an equivalent program; for instance, NASA's Path Finder would suffice. Similarly node coverage could be substituted with a stronger criterion (node coverage happens to scale well from a computational standpoint). We also envision practitioners extracting test paths from their operational profiles and/or test sets in place of `Graph-Coverage`. In short, there are many possibilites. We challenge the community to examine the effectiveness of these other techniques in place of our static approximation.

## B. Implementation

We have developed tools to compute the above testability definition for arbitrary C and Java code. We developed the tools with maximum reuse of common Linux tools in mind. We hope that practitioners will benefit from our insights into developing tools of their own.

The tools build a database mapping the source code line numbers to the control flow graph. To build the database, the tools parse the compiler's internal control flow graphs. From this database, the tools computes the testability scores by invoking `Graph-Coverage`, parsing that format, then computing (2). To get human-readable results, the tools perform a join operation to map the `Graph-Coverage` output back to source line numbers. They also generate a GraphViz [9] map.

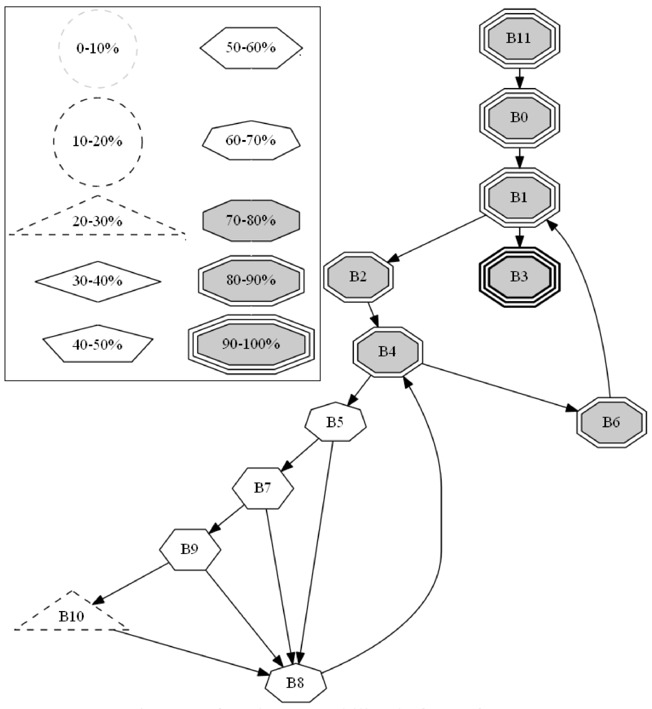

Fig. 1. A function's testability, before refactor.

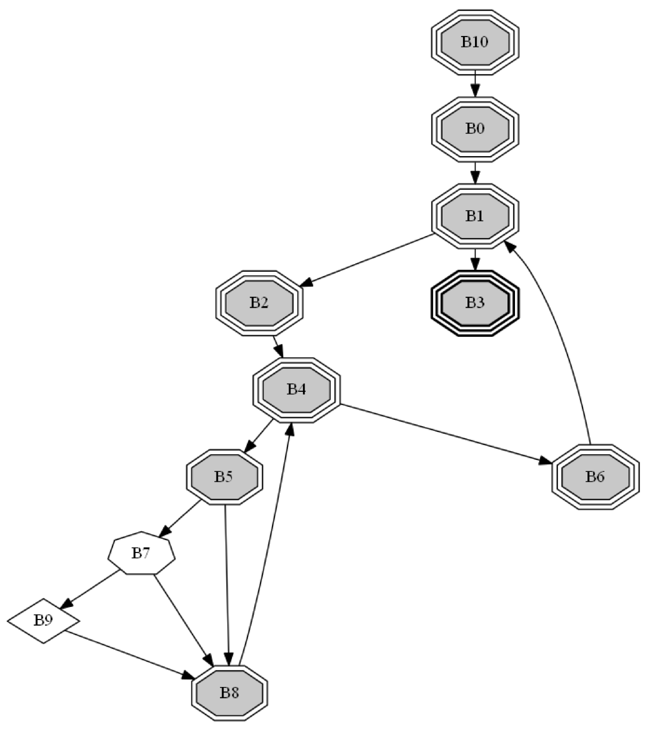

Fig. 2. A function's testability, after refactor

The tools defer the parsing of C code to the `gcc` compiler [10]. The compiler can dump the control flow graph at many stages of compilation. We decided to use the Single Static Assignment (SSA) representation [11]. The SSA dump is useful because it explicitly lists the *basic blocks* (atomic groups of lines), the edges, and the mapping from block to source lines. For very small projects, practitioners can generate the SSA dump for their own purposes using:

```
gcc -g -fdump-tree-ssa-lineno-blocks
```

Several open source projects use `configure` scripts that configure the compiler for all files. The following command (on one line) configures the compiler to create a SSA file for every file that it compiles:

```
./configure CFLAGS="-g
-fdump-tree-ssa-lineno-blocks"
```

With the SSA file available for each compiled C file, our tools simply use `awk` [12] to parse the SSA line-by-line. From the resulting database we can easily generate the graphs in the adjacency list format that `Graph-Coverage` expects.

The `gcc man` page seems to favor `-fdump-tree-cfg` and `-fdump-tree-vcg` for generating graphs [13]. We wish the SSA format was documented in greater detail on this page. We found the CFG dump inferior to the SSA dump because it does not explicitly state the edges. As for the VCG dump, we found that the VCG format comes from an early compiler pass that can very rarely contain dead code. For sharing the structural data of sensitive code with other parties, this could prove to be a useful feature. For our purposes, the dead code was not worth it.

The compiler throws out nonfunctional lines. For instance, if a function call taking many arguments is spread across multiple lines, the compiler will treat the call as if it were on only one line. This behavior is problematic for our purposes. To resolve this issue, we let lines that the compiler does not specify inherit the last known testability.

### C. Vision

Our vision is to use *some* fault proneness metric to guide automatic code refactoring. In this section, we give an example using testability for illustration purposes. We have not yet performed a longitudinal study validating this particular vision, but it is in the spirit of McCabe's ideas about “reducing” code [4].

Figure 1 contains part of the output from our Java tool: the testability of a 12 node control flow graph (the mapping from nodes to source lines is omitted). As Fig. 1 shows, node B10 is hardest to test. We envision an automatic refactoring process that would allow us to fold B10 and B9 together by moving that code into its own function. Intuitively, as Fig. 2 shows, this change would reduce the number of test paths and thus make the code overall easier to test. It should be very feasible to implement this process with a per-line metric such as testability because it precisely identifies fault-prone code. We posit that this process could reduce the fault proneness of the code by making the code less complex.

## IV. Case Study

In our case study, we examined the link between testability and fault proneness in Apache HTTP Server, otherwise known as "`httpd`" or simply "Apache." We also examined the quality of testability as a fault proneness predictor in terms of recall and precision. To our knowledge, no one has studied fault proneness on a per-line basis, so for comparison, we also compared two other metrics: McCabe cyclomatic complexity and random.

Apache is a well-known open-source HTTP server written entirely in C. Its licensing allows many commercial HTTP servers to reuse its code. We examined the Apache "trunk" SVN revision history to conduct our case study.

### *A. Research Questions*

This section states our hypotheses. We were interested in two ideas: RQ1) how the testability of Fault Prone (**FP**) code compared with Not Fault Prone (**NFP**) code, and RQ2) how well our testability measure predicted the precise location of faults.

#### *1) Difference in means (RQ1)*

Our null hypothesis ( $H_0$ ) states that there is no difference between the mean testability of FP code and the mean testability of NFP code. Our alternate hypotheses states that there is a difference, either:

- $H_1$ : FP code had lower testability than NFP code (what we posit), or
- $H_2$ : FP had higher testability than NFP code.

We reject the null hypothesis if the difference is significant within 95% confidence ( $\alpha=0.05$ ). We accept the null hypothesis only if we have at least 80% statistical power ( $\beta=0.2$ ) and the difference is not significant.

#### *2) Recall and precision (RQ2)*

To determine the quality of our predictions, we also performed a traditional fault proneness experiment by assessing the recall and precision of our estimates. We define recall, precision, and the hybrid measure F1 as:

$$\text{recall}=\frac{\text{true positives}}{\text{true positives}+\text{false negatives}} \quad (3)$$

$$\text{precision}=\frac{\text{true positives}}{\text{true positives}+\text{false positives}} \quad (4)$$

$$\text{F1}=2\cdot\frac{\text{precision}\cdot\text{recall}}{\text{precision}+\text{recall}} . \quad (5)$$

Thus our null hypothesis is that there is no difference in recall and precision between testability, McCabe cyclomatic complexity, and random. Our alternate hypothesis is that there is a difference (two-sided).

### *B. Procedure*

We examined 43 random revisions modifying C code of Apache trunk in the range [1082630,1377685]. We picked 43 based on a preliminary statistical power analysis for an overall hypothesis test. We skipped revisions correcting spelling mistakes in comments or Windows-specific code (we could not cross-compile it on our Linux system). When we picked revisions at random from the above range, we noticed most of them changed the Apache "modules" as opposed to the actual server. Historically, we observed the proportion in Apache was more even. To reduce the potential bias of using so many revisions to modules, we selected 22 revisions to the modules and 21 revisions to the server.

For each revision, we identified the C files changed by that revision. For each changed C file, we ran our tool to compute the testability of each source line. The testability scores for the changed lines formed the FP data set. The remaining scores went into the NFP data set.

To implement this procedure, we configured the SVN diff to output an `ed` [14] script, an old but easy-to-parse form of diff that gives the line numbers of changed and deleted lines with respect to the original file's line numbers. We then built the database for the original revision and queried the database for the testability scores of the changed lines. To handle added lines, we computed the testability of the added lines in reverse: we updated to the next revision, recompiled the database, and extracted the deleted line numbers in the reverse SVN diff.

Testability returns a number in [0,1], but cyclomatic complexity returns a positive integer; both metrics need thresholds defining whether a line is FP or NFP. To set thresholds in a "fair" way, we computed 11 percent ranks in [0,1], incrementing 0.1 every time. For each rank, we computed the corresponding testability and McCabe cyclomatic complexity. We used the values as our FP/NFP thresholds.

We also plotted the recall and precision from randomly ranking blocks. In theory, the recall of random choices should scale linearly from 0 to 1, while the precision of random should remain roughly flat at the overall FP sample proportion. On the graphs of recall, precision, and F1, methods that are better than random will be superlinear (above random) while methods that are worse than random will be sublinear (below random).

### *C. Results*

This section discusses the results of the comparison of testability means as well as the recall and precision.

TABLE I. Apache summary statistics

| Measure | FP | NFP |
|---|---|---|
| ***Basic blocks*** | 341 | 22638 |
| ***Mean*** | **0.34** | **0.31** |
| ***Variance*** | 0.10 | 0.12 |

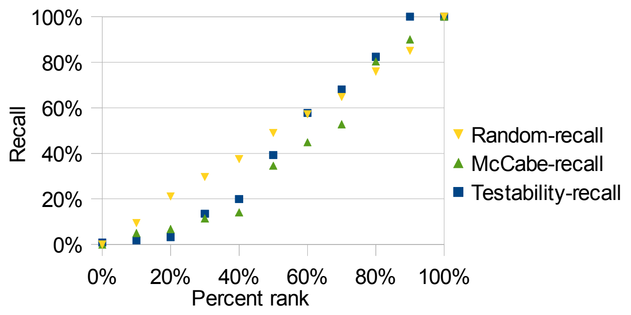

Fig. 3. Recall of testability, cyclomatic complexity, and random

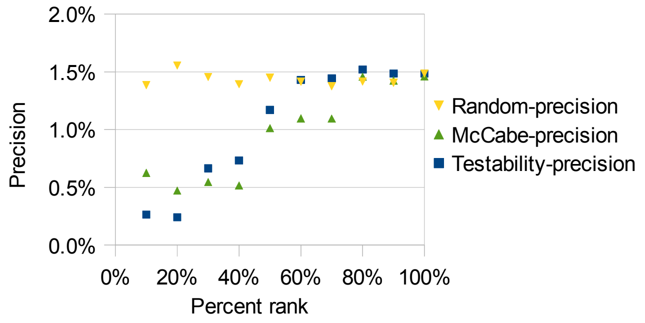

Fig. 4. Precision of testability, cyclomatic complexity, and random

TABLE II. Results of U-test between FP and NFP groups

| Measure | Value |
|---|---|
| ***p-value*** | **$1.35*10^{-6}$** |
| ***Effect size (Cohen's d)*** | 0.06 |

*1) Difference in means (RQ1)*

Table I summarizes the files present in the 43 revisions. As Table I shows, only about ~1.5% of basic blocks changed over the 43 revisions. The FP code, on average, had higher testability than the NFP code. In other words: developers found faults in easy-to-test code, supporting $H_2$. The data was not normally distributed, so to determine the significance of the difference in the means, we used the `ranksum` test in Matlab corresponding to the Mann-Whitney U-test nonparametric comparison of means.

Table II gives the results of the test. As Table II shows, the difference in the means was significantly different and exceeded our confidence threshold of 95%. The effect size, Cohen's *d*, (the difference between means normalized by a standard deviation) was only 0.06, but the difference was still significant because the data was not normally distributed. Thus we reject $H_0$ in favor of $H_2$.

To perform finer-grained testing, we applied k-means to cluster the blocks into three testability groups: low, medium, and high testability. Within each cluster, we again partitioned the blocks into FP and NFP sub-clusters and repeated our analysis. We applied the t-test to compare the FP and NFP sub-clusters. Table III summarizes our results. Within the "Low" testability cluster, the FP blocks had significantly higher testability that the NFP blocks. The effect size was moderate at 0.6428. Thus the trend of developers finding bugs in easy-to-test code was actually strongest in the Low group.

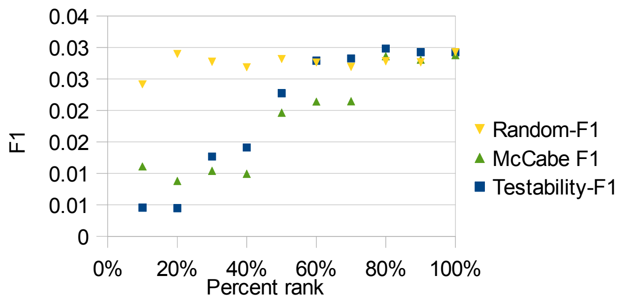

Fig. 5. F1 of testability, cyclomatic complexity, and random

TABLE III. Results of 3-means clustering

| Measure | Low | Medium | High |
|---|---|---|---|
| ***Blocks (FP)*** | 209 | 72 | 60 |
| ***Blocks (NFP)*** | 14157 | 4139 | 4342 |
| ***Mean*** | 0.0780 | 0.4487 | 0.9462 |
| ***Mean (FP)*** | **0.1215** | 0.4586 | 0.9394 |
| ***Mean (NFP)*** | **0.0772** | 0.4486 | 0.9463 |
| ***Variance (FP)*** | 0.0048 | 0.0130 | 0.0073 |
| ***Variance (NFP)*** | 0.0048 | 0.0161 | 0.0081 |
| ***p-value*** | **$4*10^{-17}$** | 0.4611 | 0.3336 |
| ***Effect size d*** | **0.6428** | 0.0788 | 0.1204 |
| ***Statistical power*** | 100.00% | **9%** | **15%** |

The medium and high testability clusters are inconclusive. The corresponding p-values in Table III lead us to not reject $H_0$. However, the statistical power is too low to actually accept $H_0$. Increasing statistical power is tricky because it is impeded by unequal sample proportions. While we could have easily looked at more revisions, we cannot convince the Apache developers to change more lines of code per revision.

*2) Recall and precision (RQ2)*

Figures 3, 4, and 5 show the recall, precision, and F1 of predicting the precise location of faults using the same 43 revisions from the first part of the study. The "Random-" plots show the effects of picking the stated percent of blocks at random. The "Testability-" plots show the effects of using the percent rank of the testability scores as an upper bound threshold. The "McCabe-" plots similarly show the effects of using the percent ranks of the cyclomatic complexity as a lower bound threshold (we order this series in reverse for comparison).

As Fig. 3 shows, testability was not significantly different than random at recall. According to a matched pair t-test, the p-value was 0.16. McCabe had significantly worse recall than testability (p-value 0.04) and random (0.01).

As Fig. 4 shows, testability did not have significantly different precision than McCabe (p-value 0.49) despite appearing slightly higher. Both metrics had significantly worse precision than random (p-values 0.048 and 0.006, respectively).

As Fig. 5 shows, testability did not have significantly higher F1 than McCabe (p-value 0.15), but had significantly worse F1 than random (p-value 0.04). McCabe was also significantly worse than random (p-value 0.03). The F1 for both methods peaked at the 80% percent rank. This rank corresponds to a cyclomatic complexity of 10 or more, which was McCabe's rule of thumb [4]. The testability at the 80% rank is 0.66, suggesting a rule of thumb that code with testability less than 0.66 needs testing.

In absolute terms, the precision of all methods examined was atrocious. Although testability significantly improved recall over McCabe, neither method was especially precise. This point leads into our threats to validity.

### D. Threats to validity

In terms of *content validity*, we did not study all revisions. While we could have studied more revisions, we could not study *all* past revisions because the older revisions require significant system modification to compile. This factor artificially caps the precision of random. Even if we could compile all revisions, the data would still be incomplete for the usual reason: the false positive code could indeed have faults that have yet to be discovered. Thus the true precision of the methods could be understated. Mitigating this threat is the fact that the data still supported McCabe's rule of thumb.

In terms of *internal validity*, there could be selection bias from using recent revisions. We tried to mitigate selection bias by being random, but a random selection across a longer time period would have been better. As said above, we could not compile very old revisions. Also, we did not model all possible effects on testability, for instance the effects of specific files or effects of the modules and server directories; we treated these as random effects.

In terms of *external validity*, we only studied Apache. While Apache is a very "real world" project, it could be that other projects, such as Eclipse, display different trends regarding testability and cyclomatic complexity. Hata et al. note that fault proneness metrics tend to perform especially well on Eclipse [3], so our results might not be directly comparable with Eclipse papers.

## V. Conclusion and Future Work

We examined recent revisions in the Apache SVN log and found, to our surprise, that developers had a tendency to find bugs in easy-to-test code. The more complicated the code became, the more evident the pattern became. This result went against our posited hypothesis that developers find bugs in hard-to-test code.

In Apache, we found the recall of testability as a fault-proneness predictor was significantly better than McCabe cyclomatic complexity when applied on a per-line basis. We found evidence confirming McCabe's "10-or-more" rule of thumb for deciding what code to test. We introduced our own rule of thumb: lines with testability 0.66 or less need to be tested.

Our challenge to the fault proneness community is to use fault proneness metrics to make specific code improvement recommendations. We hope that such recommendations will help practitioners improve fault-prone code and thus simplify their testing efforts. We discussed one possible approach: using a line-specific fault proneness metric to automatically refactor fault-prone lines out of complicated functions. We constructed testability with this approach in mind. The current function-level metrics we see in fault proneness studies, such as McCabe cyclomatic complexity, do not suffice for our purposes.

## Acknowledgment

We would like to thank Nan Li and Dr. Jeff Offutt for providing us with the latest version of their Graph Coverage tool.

## References

[1] G. Denaro, "Estimating software fault-proneness for tuning testing activities," in *Proceedings of the 22nd international conference on Software engineering*, Limerick, Ireland, 2000, pp. 704–706.

[2] "10 Dollars Bug Fix on Dilbert.com." [Online]. Available: http://search.dilbert.com/comic/10%20Dollars%20Bug%20Fix. [Accessed: 10-Sep-2012].

[3] H. Hata, O. Mizuno, and T. Kikuno, "An extension of fault-prone filtering using precise training and a dynamic threshold," in *Proceedings of the 2008 international working conference on Mining software repositories*, Leipzig, Germany, 2008, pp. 89–98.

[4] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[5] J. M. Voas, "PIE: a dynamic failure-based technique," *IEEE Transactions on Software Engineering*, vol. 18, pp. 717–727, Aug. 1992.

[6] R. S. Freedman, "Testability of software components," *IEEE Transactions on Software Engineering*, vol. 17, no. 6, pp. 553–564, Jun. 1991.

[7] N. Li, F. Li, and J. Offutt, "Better Algorithms to Minimize the Cost of Test Paths," presented at the Fifth IEEE International Conference on Software Testing, Verification and Validation, Montreal Canada, 2012.

[8] "Java Path Finder." [Online]. Available: http://babelfish.arc.nasa.gov/trac/jpf. [Accessed: 22-Jan-2012].

[9] "Home | Graphviz - Graph Visualization Software." [Online]. Available: http://www.graphviz.org/. [Accessed: 29-Mar-2011].

[10] "GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF)." [Online]. Available: http://gcc.gnu.org/. [Accessed: 10-Sep-2012].

[11] "SSA for Trees - GNU Project - Free Software Foundation (FSF)." [Online]. Available: http://gcc.gnu.org/projects/tree-ssa/. [Accessed: 10-Sep-2012].

[12] "The GNU Awk User's Guide." [Online]. Available: http://www.gnu.org/software/gawk/manual/gawk.html. [Accessed: 10-Sep-2012].

[13] "gcc(1): GNU project C/C++ compiler - Linux man page." [Online]. Available: http://linux.die.net/man/1/gcc. [Accessed: 10-Sep-2012].

[14] "ed - GNU Project - Free Software Foundation (FSF)." [Online]. Available: http://www.gnu.org/software/ed/. [Accessed: 13-Sep-2012].