Failure is a Four-Letter Word

– A Parody in Empirical Research –

Andreas Zeller[*]
Saarland University
Saarbrücken, Germany
zeller@cs.uni-saarland.de

Thomas Zimmermann
Microsoft Research
Washington, USA
tzimmer@microsoft.com

Christian Bird
Microsoft Research
Washington, USA
cbird@microsoft.com

ABSTRACT

Background: The past years have seen a surge of techniques predicting failure-prone locations based on more or less complex metrics. Few of these metrics are *actionable*, though.

Aims: This paper explores a simple, easy-to-implement method to predict and avoid failures in software systems. The IROP method links *elementary source code features* to known software failures in a lightweight, easy-to-implement fashion.

Method: We sampled the Eclipse data set mapping defects to files in three Eclipse releases. We used logistic regression to associate programmer actions with defects, tested the predictive power of the resulting classifier in terms of precision and recall, and isolated the most defect-prone actions. We also collected initial feedback on possible remedies.

Results: In our sample set, IROP correctly predicted up to 74% of the failure-prone modules, which is on par with the most elaborate predictors available. We isolated a set of four easy-to-remember recommendations, telling programmers precisely what to do to avoid errors. Initial feedback from developers suggests that these recommendations are straightforward to follow in practice.

Conclusions: With the abundance of software development data, even the simplest methods can produce "actionable" results.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics – *process metrics, product metrics;* K.3.2 [**Computers and Education**]: Computer and Information Science Education – *computer science education;* K.7.4 [**The Computing Profession**]: Professional Ethics – *codes of good practice;*

General Terms

Measurement, Experimentation

Keywords

Empirical Research, Parody

1. INTRODUCTION

In empirical software engineering, it is a long-standing observation that failures follow a *Pareto* distribution: The largest part of software defects occurs in a small fraction of software components. Therefore, research has concentrated on identifying features that correlate with the presence of software defects – features such as the number of changes, code complexity, or the

number of developers associated with a file. As elaborate as these approaches may be, they all share the same problem which we call *the cost of consequence*: If I know that a module is failure-prone because it frequently changes, should I stop changing it? If I know failures are related to complexity, should I rewrite it from scratch? Any of these measures induces a new risk – a risk which may be greater than the one originally addressed.

In this paper, we take a different approach. We predict failures from the most basic actions programmers undertake, focusing on the actions that introduce defects *as they are being made* – literally at the moment the source code is typed in. Our recommendations are *immediately actionable:* A simple visual representation associates actions with the likelihood of introducing defects – warning programmers before they might hit the wrong key. Our approach is both effective and efficient: In a case study on the Eclipse failure set, it correctly identified up to 74% of the failure-prone modules, which is on par with the most elaborate predictors available. Specifically, our contributions include:

1) A novel mechanism to associate programmer actions with software defects;

2) A predictor that is purely text-oriented, thus lightweight, real-time, easy to implement, and language-agnostic;

3) A set of easy-to-remember recommendations, validated on the well-known Eclipse dataset.

The remainder of this paper is organized as follows: We start with motivating our approach (Section 2), linking basic program features to failures. Section 3 evaluates our approach on the Eclipse bug data set, reaching new heights in accuracy. Section 4 discusses threats to validity, followed by an outline of future work in this area in Section 5.

2. THE IROP APPROACH

Empirical research has long focused on finding *abstractions* that would correlate with failures – in the hope that addressing these abstractions would also get rid of the failures. In the end, though, all these abstractions (just like software as a whole) are nothing but the product of *elementary programmer actions* such as opening files, writing tests, or running programs. To change programmer behavior for the good, we must act at an abstraction level where such change is actually feasible. (Clearly, we cannot prohibit programmers from opening files!)

Interestingly enough, it is the *lowest* abstraction layers where change becomes actionable. In the end, we can express programmer actions as a series of low-level human-computer interactions, such as moving the mouse, or typing on the keyboard. The latter

---

[*] Andreas Zeller was a visiting researcher with Microsoft Research, Washington, USA while the research leading to this paper was conducted.
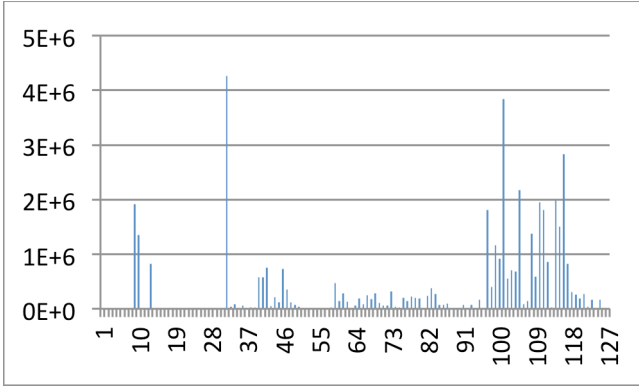
Figure 1: Character occurrences in Eclipse 2.0

style of interaction is especially interesting, as its effect is immediately reflected in the program artifacts being created. Indeed, we can interpret source code as the product of a long sequence of keystrokes, immediately visible in the program text.

One may argue at this point that this again is too much of an abstraction, as the final product (the source code) would not conserve all the editing actions that lead to it. When it comes to actionable consequences, though, treating source code as a product of keystrokes has several advantages, as we shall see later in this paper. Let us thus formulate our research hypotheses:

H1. *We can predict defects from programmer actions.*

Should H1 hold, we can test the next hypothesis:

H2. *We can isolate defect-prone programmer actions.*

These failure-correlated actions are what we call *IROP*s, which is an airline industry acronym for "irregular operation". (IROP also refers to the four most important features to avoid in source code, as detailed in Section 3.5.)

Good predictive power and actionable results lead to our final hypothesis, stating the ultimate goal of our research:

H3. *We can prevent defects by restricting programmer actions.*

3. EVALUATION

3.1 Study Subject

The key challenge for empirical research is to find appropriate data sets that would allow linking failures to program features. To encourage replication and public assessment, we selected the publicly available Eclipse bug dataset [1] [2] for our studies. It maps between 6,729 files (for Eclipse 2.0) and 10,593 files (Eclipse 3.0) to the number of pre- and post-release defects found and fixed in each file.

3.2 Independent Variables

For our investigation, we needed to establish a relation between specific actions and defects. For this purpose, we modeled a programmer action as one of 256 possible *keystrokes,* one for each 8-bit ASCII character. The result of these keystrokes is easily measured by the number of occurrences in individual source code files. Figure 1 shows the distribution of characters 1–127 across all files in Eclipse 2.0; space (ASCII code 32) is the most frequent character, followed by "e" (101), and "t" (116), which also happen to be the most frequent letters in the English language. Note that while there is a clear bias towards printable and blank 7-bit characters, there is nothing to assume that such a bias would be specific to Eclipse source code.

3.3 Dependent Variables

Our dependent variable in this setting is whether a file would be defect-prone or not. We only care for post-release defects, as these would be the ones impacting actual users. Table 1 provides descriptive statistics on these features.

Release	Total chars	Total files	Files with defects
Eclipse 2.0	44,914,520	6,728	975 (14%)
Eclipse 2.1	56,068,650	7,887	854 (11%)
Eclipse 3.0	76,193,482	10,593	1,568 (15%)

Table 1: Features of the Eclipse datasets.

3.4 Predicting Defects by Actions

We start with a standard research question, namely asking whether programmer actions predict the defect-proneness of files. For this purpose, we replicated a standard setting, training a *model* from a set of features (c, d) for each file f. Here, c would be 256-tuple denoting the occurrence counts over all 256 characters in f, and d would be a Boolean value expressing whether f has had a defect fixed in the past or not. Our null hypothesis would be:

H0. *A character distribution is not sufficient to predict defect-proneness.*

In our experiment, we used a logistic regression model, as provided by the R statistical package. Having trained the model on one of the Eclipse datasets, we used it to classify files f' in the other data sets whether they would contain defects or not. Table 2 lists the *precision* we obtained for our experiments. For instance, training the model on Eclipse 2.0 (first row) and predicting whether files would be defect-prone in Eclipse 2.1 yields a precision of 0.39 – that is, 39% of all files predicted to be defect-prone actually are defect-prone. Note that this is the *worst* of all precisions observed; on average, more than 50% of all files are correctly classified, bringing them on par with the best defect predictors.

| Training Set	Eclipse 2.0	Eclipse 2.1	Eclipse 3.0	Average
Eclipse 2.0	0.74	0.39	0.49	0.54
Eclipse 2.1	0.55	0.64	0.56	0.58
Eclipse 3.0	0.57	0.40	0.64	0.54
Average	0.62	0.47	0.56	0.55

Table 2: Precision for various training/testing combinations.

One feature we found striking was how well the model performed when used *within one release of Eclipse only*. This setting is particularly important when applying the prediction during development of a release – in a way, "training on the job". When applied within Eclipse 2.0 only, the precision is 74%, which makes this a highly useful prediction tool.

Training Set	Eclipse 2.0	Eclipse 2.1	Eclipse 3.0	Average
Eclipse 2.0	0.32	0.27	0.27	0.28
Eclipse 2.1	0.03	0.18	0.14	0.11
Eclipse 3.0	0.19	0.16	0.20	0.18
Average	0.18	0.20	0.20	0.19

Table 3: Recall for various training/testing combinations.

In terms of *recall*, our approach fares less well (Table 3), but this is a feature (or problem) shared with most defect predictors. Still, applied within Eclipse 2.0, our approach correctly identifies 32%
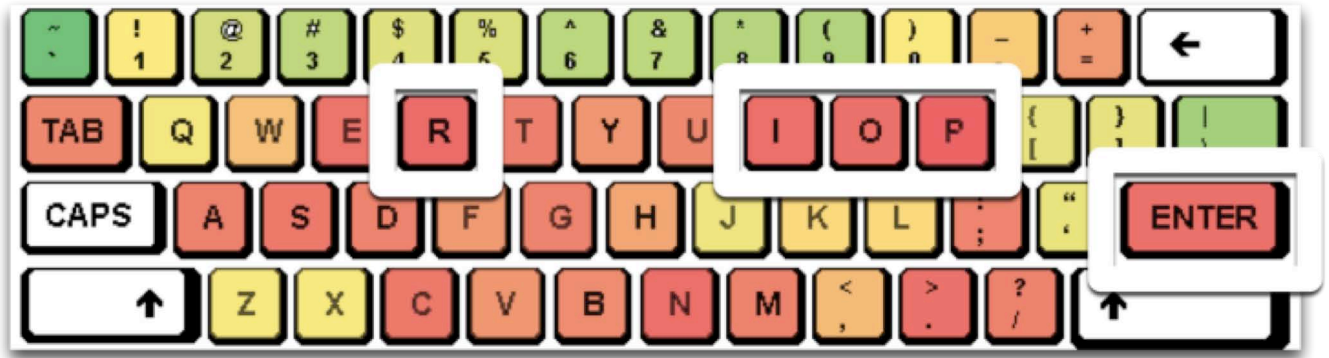
Figure 2: Color-coding keys by their defect correlation; (red = strong). The five strongest correlations are highlighted.

of the actually defective files; the average is close to 20%. Again, this rejects the null hypothesis and supports our hypothesis **H1**:

□

Programmer actions (keystrokes used to create source code) serve as excellent defect predictors, with a precision of up to 74% and a recall of up to 32%.

3.5 Programmer Actions and Defects

Now that we know how to predict defects, can we actually prevent them? Of course, we could focus quality assurance on those files predicted as most defect-prone. But are there also *constructive* ways to avoid these defects? Is there a *general rule* to learn?

For this purpose, let us now focus on **H2**: Is there a correlation between individual actions (= keystrokes) and defects? For this purpose, we would search for correlations between the count of the 256 characters and the overall post-defect count per file; our null hypothesis would be:

H0. *There is no correlation between character distribution and defect-proneness.*

After a number of preliminary experiments, we focused on the Eclipse 3.0 dataset. It is well known that most metrics of software do not follow a normal distribution and our measures of keystrokes are no exception. The distributions of characters appear to have an exponential rather than a power-law character. Nonetheless, due to the heavily skewed distribution, we used a standard non-parametric approach with the Spearman rank correlation. Of course, with so many metrics (one for each character), we run the risk of identifying spurious correlations, and we thus employed p-value adjustment using Benjamini-Hochberg p-value correction [3] to deal with this multiple hypothesis testing. In order to be conservative in our findings and avoid Type I errors, we used a p-value cutoff of $\alpha = 0.01$ for statistical significance [4]. Even after taking these rigorous steps, all letters and digits showed a statistically significant positive correlation with failures.

For the non-printable characters, this correlation is strongest for the newline character (0.34). The correlation with newline characters is not surprising: given a constant defect density, a file with more lines would be assumed to also have more defects. For the printable characters, though, we observed the highest correlation for the lower-case letters "i" (0.34), "r" (0.34), "o" (0.34), and "p" (0.35) – in other words, the more of these letters one would have in a file, the higher the defect count. This is the more interesting as these letters do *not* rank in the most frequently used English letters; this is also in sharp contrast to characters such as "%" (0.06) or the uppercase "Z" (0.19). Figure 3 lists the correlations for the individual lower-case letters.
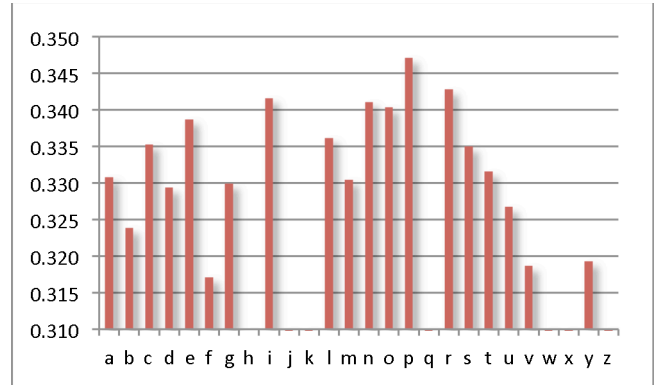
Figure 3: Defect correlation for the 26 lower-case letters.

This high correlation for the specific letters "i" (0.34), "r" (0.34), "o" (0.34), and "p" (0.35) came as a huge surprise to us; it is these specific letters that named our approach IROP. All reported correlations are statistically significant ($p = 0.01$), refuting H0 and confirming our hypothesis **H2**.

□

Our results show a strong correlation between specific programmer actions (keystrokes I, R, O, and P) and defects.

3.6 Preventing Defects

Correlations like the above give way to immediate action. Our first idea was to encode the defect likelihood as *colors* into the keyboard (Figure 2), such that programmers would be aware of the risk immediately when undertaking the specific action.

However, such an encoding on the keyboard would not impact professional programmers, in particular touch typists. Therefore, we constructed a special keyboard that would make it harder for programmers to undertake defect-prone actions (Figure 4). Note how the four letters of failure are conveniently removed, which forces programmers to rethink their actions and to search for alternatives.[1]

We deployed this keyboard to three Microsoft interns in our group to carefully monitor its effect on defect reduction. It quickly

---

[1] We also explored removing the "Enter" key, but experienced that this led to a sharp increase in the number of defects per line as well as a drop in productivity (measured as LOC/day). These effects will be explored in future research.

Figure 4: Avoiding risky actions with the IROP keyboard

turned out that getting rid of the four letters of failure would not be an easy task. While our test subjects could easily avoid "i", "r", "o", and "p" in their identifiers, the largest problem would be *keywords* in programming languages. Our interns quickly came up with appropriate replacements, though. C# code such as

```
if (p != null)
    { int i; while (p[i] < 0) i++; return i; }
```

becomes

```
when (q != null)
    { num n; as (q[n] < 0) n++; handback n; }
```

which is just as readable as before. Such transformations can easily be performed automatically even on million-line programs; furthermore, they are 100% semantics-preserving, thus ensuring no unintended consequences.

Getting rid of "i", "r", "o", and "p" is now part of programming culture, as one of our interns remarks:

We can shun these set majuscules, and the text stays just as swell as antecedently. Let us just ban them!

Note how our test subject already avoids the four letters of failure in his statement; of course, it was typed on the IROP keyboard.
□

Programmers can easily memorize the IROP principles and adapt their work habits to proactively prevent failures.

4. THREATS TO VALIDITY

The results of our experiment are subject to the following threats to validity:

Threats to external validity concern our ability to generalize the results of our study. Our findings are based on more than 177 million of individual characters (see Figure 1) and thousands of individual defects, making this one of the largest empirical studies on defect prediction ever conducted. However, we would not advise to generalize the results beyond the C/C++/C#/Java family of languages, due to different keywords. Likewise, source code using non-English identifiers or comments needs a separate investigation, as detailed in this paper.

Threats to internal validity concern our ability to draw conclusions between our independent and dependent variables. Due to the high number of instances, we could test all correlations to be

significant. We used well-established tools and techniques to produce all results, using only a few lines of own code which were trivial to validate.

Threats to construct validity concern the appropriateness of our measures for capturing our dependent variables. By definition, source code is always produced by humans, and all of it is input as characters; indeed, characters as we find them in the source code are the very source of all defects. In this work, the deliberate absence of abstractions not only completely eliminates threats to construct validity – it is also the reason why the IROP approach is so effective.

5. FUTURE WORK

With the abundance of software development data, even the simplest methods can produce actionable results. IROP is not only straight-forward to implement; it also produces recommendations that are easy to understand and easy to follow. Besides general refinement and improvement of the technique, our future work will focus on the following issues:

Automation. As discussed in Section 3.6, we must not only prevent new errors, but also *refactor existing code* to avoid defect sources. We are currently working on IDE plug-ins that will conduct thesaurus-based renamings automatically at the moment the code is loaded, providing synonyms without the four letters of failure; where this is not possible, the letters shall simply be removed. First betas for the Eclse and Seeable Study IDEs shall be made available through the web page of ENGAGEMENT 2011 (formerly known as PROMISE 2011).

Abstraction. As with any concrete symptom, we must carefully check whether there would be a common *abstraction* that could explain these effects. One hypothesis is that programmers would subconsciously associate these four letters with negative terms: "Failure", "mistake", "error", "problem", "bug report" all contain "i", "r", "o", and "p"; whereas "success" and "fame" do not. We plan to run controlled studies to validate these "a priori" hypotheses.

Generalization. As detailed in Section 4, our findings are based on English code and comments only so far. We are currently exploring Russian source code and whether a high abundance of the IROP equivalent characters ИРОП would also correlate with a high number of defects; we are very, very confident they will.

6. WHY ALL THIS IS WRONG

While reading through this paper, you may (and actually should) now have come to the conclusion that all of this is nonsense: Of course, none of us wants to eradicate individual letters from programs. But where does the nonsense actually begin? And is it confined to this paper alone? Unfortunately, it is not. The widespread availability of empirical data in software engineering has brought an explosion of findings – many of them substantial, but some of them banal at best, and misleading at worst.

With this paper, we have tried to replicate a number of blunders that we have found in papers of researchers; some such papers actually are submitted to respected venues. The numbers and correlations reported are all true findings from the Eclipse dataset, and correct to the best of our knowledge. However, it is the *interpretation* of the results that is plain wrong. Before we start discussing our deliberate blunders, maybe you'd like to go back to the previous sections and search for them yourself.

6.1 High-level Issues

Let us start with a number of high-level issues demonstrated in this paper. In contrast to our study, there is a wealth of high quality work coming out of our community. We highlight a few examples of *good* empirical research along the way.

Correlations do not imply causations. Our paper implies that a high correlation is an important discovery. But any sufficiently large data set will contain lots of correlated data, and it is easy to find them. There are additional requirements, however (see [5], pages 80-81 for details). To show causation, one also needs to show that changing the cause also changes the effect (which of course we did not), and one needs a substantial *theory* to explain causation (for which our "abstraction" hypothesis in Section 5 is only a ridiculous surrogate). For such a theory, one needs *domain knowledge.* And such knowledge cannot come from data alone; you must investigate and understand the stories behind the data. (For a detailed discussion on the importance of theories in Software Engineering, see [6]).

We invite the reader to examine an empirical study that goes far beyond mere correlation. In their paper on "Developer Fluency", Zhou and Mockus [7] carefully constructed a theory of developer knowledge acquisition *based on prior literature* and refined it by gathering qualitative data from interviews. Based on this theory they developed, validated, and triangulated measures such as time and task difficulty that showed a relationship with productivity. They also showed how differences between projects (yes, they looked at more than one!) accounted for differences in findings.

Do not confuse causes and symptoms. Our paper argues that it may be worth looking at low-level features rather than abstractions. While this may be true for initial exploratory studies, it is crucial to reason whether there would be a common abstraction that would explain the effects observed. In a file, it is pretty much obvious that any occurrence of letters would correlate with its size; and it is not very surprising that given a constant defect density, the larger the file, the more defects it will have. *This* would be the abstraction to look at and the lowest baseline to compare against.

Focusing on individual letters is actually an issue of *construct validity*: The construct we chose for modeling programmer actions is plain inadequate. We recommend the book by Shadish et al. on experimental design and causal inference for a detailed discussion [8].

Few findings generalize. Even if you could empirically show that there is a strong correlation in a single data set (a correlation of 0.35 is not strong), it is unlikely that you will find the same correlation in another project, or even another release of the same project. Note that while we pretend to look at multiple releases of the Eclipse project, our "IROP" finding is based on the 3.0 release alone. In Eclipse 2.0, the "IROP" principle becomes the "Namp" principle; in Eclipse 2.1, it becomes the "Nogl" principle.

As an example of a research that provides value without generalization, in 2007 Hindle et al. [9] undertook an investigation to understand the rationale behind large commits and determine if anything could be learned from studying them. They studied large commits from nine software projects, and while they identified a number of reasons behind large commits, they found that the reasons varied dramatically across projects. Rather than try to force a generalized conclusion, they reported their findings and project-specific, but still valuable, insights.

Beware of cherry-picking. So, why did we name our "principle" IROP and not "Namp" or "Nogl"? This is hinted at with the words "After a number of preliminary experiments, we focused on the Eclipse 3.0 dataset." What we actually did was that we looked at all three releases and picked the one that fit us best, conveniently *suppressing* the differing results. Such suppression is a no-go: All relevant findings must be reported; and if you have findings that contradict your theory, well – there goes your theory.

Beware of fraud. Of course, one can always suppress inconvenient findings without even hinting at them. This is why any choice or influence from the researchers must be carefully justified and questioned – a simple "We selected five bugs to illustrate our approach" will not yield any averages, sums, or generalization; it is a mere proof that the technique *can* work. But whether this reflects a property of the technique or a property of the example must be carefully evaluated by the authors in the first place. The good news is that the most striking results will eventually be replicated and possibly refuted.

Threats should help understanding. All empirical research should point out threats to validity. No empirical study is perfect and reviewers shouldn't expect such. These threats highlight possible issues such as contexts that are not represented and in which the results are unlikely to hold and reasons that measurement may have error. They aid the reader by creating a lens through which the results can be viewed and interpreted. Note the tone of Section 5. Somehow we are able to overcome all threats by arguing ways that we did everything perfectly. There is no mention that our study is of only one software project, some characters are simply unavoidable, or causation may flow in the opposite direction. It is tempting to use this section as a way to refute any possible criticisms, but such an approach should be used judiciously and not to mask informative limitations.

Machine learning works. The past six years of mining software archives have impressively demonstrated that one can train a machine learner on tuples of features and failures, and use these very learners to successfully predict future failures. The problem for researchers is that the actionable features – that is, features that suggest direct corrective action such as changes, complexity, or test coverage – have all been studied already. The good news is that any feature that characterizes a component may be sufficient for not-too-bad results – even if it is just a distribution of characters in the source code. Again, a comparison with the state of the art (not just a simple straw man) is required to demonstrate improvement.

In their paper on defect predictors, Menzies et al. [10] showed that by carefully considering both the set of features used for prediction *and* the type of learner, better prediction performance could be achieved than prior approaches (which did not evaluate different learners). Their study is a comprehensive comparison of Naive Bayes to other learners that had been in vogue at that time and we refer the reader to this work as an example of careful use of machine learning.

Make findings actionable. As (correctly) stated in the introduction, an empirical finding is the more valuable the more *actionable* it is. What is the consequence of this result? Should I change things? How? What is the risk of this change? Your empirical finding need not provide answers to all these questions. But it should convey an idea of its potential implications.

As an example of actionable empirical results, see the work of Ramasubbu and Balan on process choice [11]. They evaluated the results of software process choice in 112 software projects and found a link with a number of measures of performance. They were able to identify five attributes of a software project, such as team size, estimated effort, and extent of client involvement, that can be used at the beginning of a project to decide if a standard "plan-driven" approach should be supplanted by a non-standard agile process. Such results are immediately useful to software practitioners.

Fix causes, not symptoms. Being non-actionable may still be better than suggesting the *wrong* actions. Complexity metrics, for instance, stipulate that specific parts of the code may be problematic. But then, it is trivial to rearrange the code (in a "100% semantics-preserving" way) to satisfy all the metrics. This helps as much as removing "i", "r", "o", and "p" from your program. In most cases, what you need instead is project-specific actions, including empirical investigations on the features that correlate with failure, and a careful search for the actionable causes and abstractions behind these features. Every project is different, and the project-specific issues far outweigh general "textbook" issues.

Get real. The new abundance of ground truth has had several beneficial effects on software engineering research: Automated tools are routinely validated on real code; hypotheses can be backed by real process data. Far too frequently though (and the authors plead guilty as charged) do we rely on data results alone and declare improvements on benchmarks as "successes". What is missing is *grounding in practice:* What do developers think about your result? Is it applicable in their context? How much would it help them in their daily work?

To get a starting point on what developers actually need, let us recommend the study by Ko et al. [12]; we leave it to you to assess which of the current research matches these needs.

6.2 Issues in Detail

Throughout this paper, we undertook our best efforts to sell our results. This is common practice; indeed, good writing is absolutely necessary to get papers accepted at the best publication venues. However, good writing is a double-edged sword. You can use it to make your presentation clear, precise, and easy to follow. But you can also use it to manipulate the reader to gloss over findings that would not withstand careful scrutiny. For instance, did you notice how

- We deliberately used "programmer actions" as a high-level substitute for the much more banal "characters in a file"?
- Tables 2 and 3 (precision and recall) also contain entries in which the same releases are being used both to train and to

test? Any such predictor, of course, would fare well; these entries conveniently beef up the average.
- We justified our choice of Eclipse "to encourage replication". This paper does nothing to encourage replication, as it does not provide any new data or artifact; the choice of Eclipse remains unjustified.
- We consistently came up with an immediate interpretation for each and every number, such as the "high" correlations for individual letters?
- We compared correlations against each other, without further testing the resulting hypotheses?
- The Y axis in Figure 2 is conveniently set up to suggest large differences, which actually are very minor?
- We (at best) have anecdotal support for hypothesis H3?
- We conveniently avoided any comparison against a baseline? Or any related work?
- Figure 2 actually shows that "n" has a higher correlation than "o", making this the INRP principle?[2]
- Our threats to validity carefully avoid all these central problems? (And what would the "real" threats to validity look like?)

Many more of such manipulations can be found in the classic "How to lie with statistics" [13]. Although mainly aimed at false advertising, many of the discussed advertising patterns can equally be applied to scientific papers.

6.3 What was Right

Like most parodies, this paper is grounded in real facts. The motivation on making findings actionable (Section 1) has a grain of truth in it. All numbers as reported are correct, and it is indeed possible to predict the defect-proneness of files using character counts as prediction features. (Note how little this demands in terms of implementation.) The correlations listed are also all true to the best of our knowledge; only Figure 4 and the "intern" statements are pure fabrications. In terms of results, this paper satisfies all the principles of serious research; it is the construct, the interpretation and the consequences that are way over the top.

6.4 Consequences

Any of these issues listed in this section could easily be picked up by an experienced reviewer. Unfortunately, real blunders are not always as obvious as in this paper; and in many cases, even the authors themselves do not know about their blunders. This is why both authors and reviewers need to be aware about such issues.

As it comes to writing, we as readers and reviewers must be aware of possible manipulations. We must carefully check the numbers and question the author's interpretations – and authors must present their findings in a way that eases and allows independent interpretation of the results. As a community, we have come a long way over the past 30 years (consider that other empirical fields such as sociology and economics have had centuries to mature!) but our empirical standards have room to improve, and we must favor inconvenient honesty over slick storylines.

7. CONCLUSION

The aim of this paper is different from what one would find in "regular" scientific publications – rather than adding new findings

---

[2] Given that all these letters have roughly the same correlation, any combination of INROP would have been fine; IROP simply made the best meme. We leave it to the imagination of the reader what the paper would look like, had we settled on PORN.

to the body of knowledge, it uses trivial "findings" to demonstrate a number of blunders in research that may be undetected by the casual reader or reviewer. With this paper, we hope to have raised awareness for these blunders.

The target audience of this paper is not so much established researchers but rather students confronted with empirical research. To facilitate classroom usage, we make available all the scripts and data for this "research" at

http://www.st.cs.uni-saarland.de/softevo/irop/

This also includes the original Word file for this paper, such that instructors may use the material (stripped of all hints and solutions) as a student exercise for critical assessment.[3] In the end, it is our hope that IROP will catch as a short meme for bad research to avoid.

8. REFERENCES

[1] Zeller, A., Zimmermann, T., Premraj, R., Just, S., and Schröter, A. *Eclipse bug data 2.0a*. 2007.

[2] Zimmermann, T., Premraj, R., and Zeller, A. Predicting Defects for Eclipse [Revised for Dataset Version 2.0a]. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering* (Minneapolis, MN, USA, 2007), ACM.

[3] Benjamini, Y. and Hochberg, Y. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, 57 (1995), 289--300.

[4] Dowdy, S., Wearden, S., and Chilko, D. *Statistics for research*. John Wiley \& Sons, 2004.

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

[6] Hannay, J.E., Sjoberg, D.I.K., and Dyba, T. A Systematic Review of Theory Use in Software Engineering Experiments. *IEEE Transactions on Software Engineering*, 33, 2 (February 2007), 87-107.

[7] Zhou, M. and Mockus, A. Developer fluency: Achieving true mastery in software projects. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering* (2010), 137--146.

[8] Shadish, W., Cook, T., and Campbell, D. *Experimental & Quasi-Experimental Designs for Generalized Causal Inference*. Houghon Mifflin, Boston, 2002.

[9] Hindle, A., Germán, D.M., and Holt, R.C. What do large commits tell us? A taxonomical study of large commits. In *Mining Software Repositories* (2008).

[10] Menzies, T., Greenwald, J., and Frank, A. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering* (2007).

[11] Ramasubbu, N. and Balan, R.K. The impact of process choice in high maturity environments: An empirical analysis. In *International Conference on Software Engineering* (2009).

[12] Ko, A.J., DeLine, R., and Venolia, G. Information Needs in Collocated Software Development Teams. In *International Conference on Software Engineering (ICSE)* (Minneapolis, MN, 2007), 344-353.

[13] Huff, D. *How to lie with statistics*. Norton, New York, 1954.

[14] Munroe, R. Significant. *xkcd*, http://xkcd.com/882/ (Accessed May 1, 2011).

---

[3] This "exercise" variant also has the authors stripped.