Evolving Defect "Folklore": A Cross-Study Analysis of Software Defect Behavior

Victor Basili[1] and Forrest Shull[2]

[1] Dept. of Computer Science, University of Maryland,
College Park, MD, 20742, USA
basili@cs.umd.edu
[2] Fraunhofer Center - Maryland, 4321 Hartwick Road,
Suite 500, College Park, MD, 20740, USA
fshull@fc-md.umd.edu

Abstract. Answering "macro-process" research issues – which require understanding how development processes fit or do not fit in different organizational systems and environments – requires families of related studies. While there are many sources of variation between development contexts, it is not clear *a priori* what specific variables influence the effectiveness of a process in a given context. These variables can only be discovered opportunistically, by comparing process effects from different environments and analyzing points of difference.

In this paper, we illustrate this approach and the conclusions that can be drawn by presenting a family of studies on the subject of software defects and their behaviors – a key phenomenon for understanding macro-process issues. Specifically, we identify common "folklore," i.e. widely accepted heuristics concerning how defects behave, and then build up a body of knowledge from empirical studies to refine the heuristics with information concerning the conditions under which they do and do not hold.

1 Introduction

Answering "macro-process" research issues – specifically, being able to make statements about the effectiveness of processes in different contexts, and understanding how processes fit or do not fit in different organizational systems and with different organizational characteristics – requires families of related studies. This is true because of two difficulties inherent in software process research:

- It is clear that there are many sources of variation between one development context and another;
- It's not clear *a priori* what specific variables influence the effectiveness of a process in a given context.

That is, we expect it to be an almost impossible task to predict ahead of time what factors are likely to crucially affect the results of applying a process in one environment or another: for example, the motivation of the practitioners, their experience/skill level with various tasks, the various business goals of the organization. Yet we know that these variables do exist and we are able to reason about their influence if we work bottom-up, that is, starting with the observation of process effectiveness in various environments and identifying the possible causes of discrepancies.

For example, one study provided some indications that the application of a particular software inspection process was influenced by the experience of the developers applying it: novice inspectors seemed to gain some improvement from the new technique while experts seemed to fall back on their own, proven practices [1]. This effect was traced to a particular context variable in this study, the time limit given for the inspection: Since the participants felt pressured to get the inspection completed in time, experts fell back on their own techniques rather than try to deal with the learning curve. In a context where the time limit was open-ended or subjects were more motivated to learn the new process, it is impossible to say whether the same effect would still have been observed. Such unexpected inter-relationships between variables are always a possibility in software process research.

For this reason, we have argued [2] that knowledge at the macro-process level must be built from families of studies, in which related studies are run within similar contexts as well as very different ones. At one level this ensures that conclusions are verified and false conclusions are not drawn due to problems or idiosyncrasies with any one study. For drawing macro-process conclusions, however, it also allows the space of context variables to be explored *opportunistically*; conclusions about influencing factors are drawn bottom-up, by understanding the context similarities and differences that may have caused differences in process effectiveness.

Multiple authors have discussed the idea of software replication, that is, how to design related studies so as to document as precisely as possible the values of likely context variables and be able to compare with those observed in new studies [e.g. 3, 4, 5, 6]. While such a top-down approach is important, given the overwhelming number of potential context variables – including differences in developer experience and motivation, in development tools and approaches, in other processes used, in business goals (e.g. high quality products vs. fastest time to market) – we argue that a bottom-up approach is also necessary, in which results from multiple individual studies can be fitted together after the fact as appropriate. Such a bottom-up approach is necessary for enabling recommendations to be made about process effectiveness in context.

To make such a bottom-up approach work, it is necessary to have an overall framework that allows the relationships among individual studies to be understood so that data can be accumulated and variations in effectiveness determined. Such a framework allows independent researchers to relate the results of their studies more easily to the growing body of knowledge so that macro-process conclusions can be drawn.

In this paper, we illustrate this approach and the conclusions that can be drawn by presenting a family of studies on the subject of software defects and their behaviors – a key phenomenon for understanding macro-process issues. Specifically, we identify some common "folklore," i.e. widely accepted heuristics concerning how defects behave, and then build up a body of knowledge to show whether empirical results can confirm whether such heuristics are accurate and under which conditions.

2 Software Defects

The comparison of lessons learned about software defects between studies has been complicated by the fact that there are multiple taxonomies of defects that have been proposed over time (e.g. [7], [8], [9]). Thus building up a body of knowledge on defects and their behaviors is made even more complicated: Not only do studies in different contexts have many different sources of context variation, which are impossible to identify ahead of time as discussed above, but they may have additionally used different vocabulary to describe similar types of defects.

To make sense of the knowledge learned about defects across multiple contexts and taxonomies, we need to go to a higher level up abstraction, to what we call "folklore." Folklore in this context refers to the informal, subjective lessons learned by developers based on experience. Due to the well-documented fact that human subjective experience is not always the best basis for drawing generalizable lessons, as well as the fact that experiences in one context may not always generalize to others, it is important to use such lore to formulate testable hypotheses that can then be subjected to more formal scrutiny. In this way, folklore is one source of information that should be used to focus new empirical studies in high-payoff areas. The hope is that the basic heuristics encoded in folklore reflect such basic knowledge about software phenomena that they are relatively insensitive to the variations in the precise definition of defect.

To facilitate the comparison across studies, in this paper we will use the IEEE definitions [10] for defects and related phenomena:

- Error: a defect in the human thought process made while trying to understand given information, to solve problems, or to use methods and tools;
- Fault: a concrete manifestation of errors within the software (note that one error may cause several faults and various errors may cause identical faults);
- Failure: a departure of the operational software system behavior from users' expected requirements (a particular failure may be caused by several faults and some faults may never cause a failure).

As examples of folklore about software defect behaviors, we introduce the following heuristics and rationales:

- *The vast majority of defects are interface defects.* This heuristic describes the common belief that implementing individual modules with clearly defined functional requirements is rather straightforward. Instead, the majority of defects is believed to come at the interfaces of such modules, that is, getting them to work together to achieve higher-level functionality.
- *Applying more sophisticated programming languages can eliminate a significant number of defects, but not all.* Another way of saying this is that most implementation defects are believed to come from coding mistakes that could be minimized by better programming languages, which would reduce the likelihood of developers making such mistakes. This is consistent with the belief that most code defects are introduced because of the complexity of solving the problem on the computer, not from problems in analyzing the solution to be produced in the first place.

- *Small modules (say, modules with fewer than fifty LOC) are the least defect prone.* It is assumed that breaking functionality down into the smallest coherent pieces, and implementing each in a separate module, minimizes the number of defects introduced. Said another way, larger modules are assumed to have on average more complexity, which leads them to be more error-prone.
- *If you are not sure what to do – do something and fix it later.* As a general implementation strategy, it is assumed that the effort to modify code is not prohibitive, so it is viable to implement functionality by adding code, testing it, and perfecting the implementation over time.
- *There are patterns in the defect classes found in projects within a particular organization.* This suggests that there are problems common to the organization and application. Thus collecting data for a particular environment will allow the organization to identify opportunities for improvement within that organization.

3 Drawing Conclusions Bottom-Up

In this section, we give a brief overview of a collection of datasets that provide partial evidence addressing the folklore introduced in Section 2. As was emphasized in our discussion of building up bodies of knowledge from families of studies, each study in the family need not be a "strict" replication of one another, with the same overall design and data collection [2].

Also as proposed in [2], we use the Goal – Question – Metric paradigm (GQM) to provide the framework that relates studies within the family to one another. The GQM requires explicit identification of an *object of study* as well as a *focus* for the study (i.e. a model of how the object of study is being characterized or evaluated). Specifying both the object and focus of study helps to make similarities and differences among studies explicit. All of the studies which produced data included in this collection, whatever their specific goals, all have GQMs which at a high level of abstraction have the same form:

> Analyze **software defects** in order to **characterize** them with respect to **various classification schemes** from the point of view of the **knowledge builder** in the **development context in which they were generated**.

As a first pass for demonstration purposes, the datasets in our collection are ones the authors are very familiar with. They include defect data from:

- Endres75: A new release of an operating system, where "defect" was defined as any code fault that caused a failure during system testing. (Faults generated from unit or integration testing were not included.) [11]
- Weiss79: A simulator of various computer architectures, in which all defects were found due to failures reported during the first year of operations after system delivery. [12]
- Basili/Weiss81: The development of an on-board flight control program for a new aircraft. Defects were defined as fixes necessary to the requirements

document, within a 15-month period after the requirements were baselined. (Since the defects were not tracked over the entire lifecycle of the project, this cannot be taken as a complete set of requirements defects.) The problems with the document were found during reviews as well as when it was used as a basis for design. [13]

- Basili/Perricone84: A system designed for satellite planning studies at NASA. Defect data was collected starting with the baselining of the code through the three years of maintenance. [14]
- Mashiko/Basili97: A set of four projects dealing with communication software. [7]
- Weiss/Basili85: A set of three projects dealing with ground support software for satellites. [15]
- Selby/Basili91: A single release of a code library tool. [16]

4 Drawing Conclusions Across Studies

Results from abstracting up across our various data sets show that not all of the folklore was an accurate reflection of software development realities:

The majority of defects are not interface-related. Five of our datasets had collected enough information about the defects to categorize them somehow as interface or non-interface related. The Endres75 dataset defined interface-related issues as any issue that required a change to more than one module in order to fix. Defined this way, the clear majority of defects (85% of the entries in the dataset) were non-interface, i.e. required changes to only one module.

Using a similar definition, the other datasets were in agreement. In the Weiss79 dataset, 94% of defects were non-interface; in Basili/Weiss81 dataset it was 85%. Concerning the Basili/Weiss85 dataset, although an absolute number is not given, the statement is made that "interface errors are not especially troublesome."

However, [14] offered a second definition of an interface defect: a defect is an interface defect if one has to examine more than one module to understand how to fix the defect. Thus even if only one module has to be changed, it can still be an interface defect. Using this definition of interface defect, 39% of the defect could have been classified as interface defects, and these interface defects were the largest single category of defects – 39% of faults involved interface. Using a similar definition of interface defects, [7] report a similar number of interface defects – 40%. Therefore, it can be concluded that across all datasets, although the vast majority of changes made to fix a defect were made to only one module, the need to examine more than one module in order to make a fix was a common problem, even if it did not involve the majority of defects.

Where the datasets showed some disagreement was on the subject of how expensive it was to correct the interface-related defects. Two datasets collected information sufficient to address this issue: In Weiss79, interface design defects were relatively inexpensive. However, in Basili/Weiss81 interface defects took more effort to fix. Since the Basili/Weiss81 defects were collected in the requirements phase, while Weiss79 were defects found in operation, interface defects may simply be more difficult to repair at the requirements level compared to other types.

Defects that could be addressed by better programming languages account for a significant portion of defects, but less than half. Our datasets also contained information about defect categories that can shed some light on what type of misunderstanding on the part of developers caused the defect to enter the system in the first place.

In the Endres75 set, approximately half of the defects (46%) originated due to misunderstanding the problem to be solved or potential solutions. A further 16% were related to textual/clerical mistakes or to not following standards. Thus only 38% of defects could have been avoided had improved programming techniques been used.

In Weiss79, only 31% of the defects were related to the implementation. The remainder were related to requirements, design, or clerical issues.

In Basili/Weiss81, the requirements document contained more defects related to the correctness or completeness of the solution (80%) than with the way it was represented in the given notation (18%). Thus, although in a different lifecycle phase, these results can be taken to agree with the earlier datasets in the sense that the primary cause of defects was misunderstanding of the problem to be solved.

Small modules are no less error-prone than large modules. Two datasets (Endres75 and Basili/Perricone84) traced the defects recorded back to the modules in which they were found, and the size of those modules. In the Endres75 dataset the defect rate (i.e. the number of defects per module divided by the size in LOC of the module) is no different for large modules than for small ones. The Basili/Perricone84 data showed the counter-intuitive result that larger modules, within limits, may even be less fault prone.

"Do something and fix it later" is not always a safe strategy. In Selby/Basili91, it was noted that during design and code review, the total time to correct a fault (identify and fix) of omission was less than the time to correct faults of commission. This result was surprising, given the folklore. Mashiko/Basili97 supports this conclusion when one considers all faults. However, when one limits the faults to those reported by the customer, the results are not consistent, i.e., faults of omission, after delivery of the system, are more expensive to fix than faults of commission when considering customer reported faults. It seems that the context here makes the difference and examining that context offers the opportunity for some insight. During development faults of omission tend to be smaller parts of the system, thus it is better to mark the spot where there is a concern, minimizing the time to identify the fault, and define the correct solution later. However, once a system is delivered, the faults of omission may take on a different flavor, i.e. an omission fault might represent a complex functional capability that the customer assumed was part of the system.

Both the Weiss79 and the Basili/Weiss81 datasets contained information about the time-to-fix needed for defects in an environment where the customers' needs were well understood. For the faults found after delivery in Basili/Weiss81, the effort required to make changes was in most cases relatively small; 68% of defects were repaired in less than one hour. In fixes required to the requirements document described in Weiss79, 84% of defects were fixed in less than a few hours each. However, it should be noted that in both cases a small minority took an exceptionally long time to repair; in Weiss79, for example, 1% of the defects took more than a few days to repair.

It should also be noted that the Basili/Perricone84 study concluded that it's more expensive to repair reused modules than ones developed from scratch. So, in some cases, it may be more cost-effective to try out a solution and then throw away the early prototype, rather than try to continue to modify the early version of solutions until they work correctly.

Patterns exist in defect classes found in projects within a particular organization. The Basili/Weiss85 dataset was used to identify patterns in the change and defect history of projects developed in NASA’s Software Engineering Laboratory. For example, most defects in the development of ground support systems were due to problems in the design and development of single components. This was due in part to the fact that systems were being developed by experienced developers and the single components were coded by novice programmers. It should be noted that the Basili/Perricone85 data showed that for a different application, in the same environment, a majority of the defects were due to requirements and functional specification. The Mashiko/Basili97 data also exhibited a pattern in the defect classes, though this pattern was quite different. Thus, there are patterns that can be detected within a given context, although these patterns will not hold from one environment to another.

5 Summary

Having looked at a collection of datasets and abstracted up a set of conclusions on specific topics, we should also examine what kinds of general lessons learned we have found about cross-study conclusions. We feel that the work described in this paper demonstrates that:

- *There is value in multiple studies for both supporting and not supporting hypotheses.* There are several instances above where the conclusions from multiple datasets all point in the same direction, thus making the overall conclusion much stronger than if it came from any single study in isolation. And, in several important instances, the results from additional studies identify important caveats by examining processes in new environments.
- *Care must be taken to make sure that the objects of the comparison are actually like things that can support the conclusions being drawn.* For example, although all of the datasets described in this paper contain defect data, it was necessary to know in each study the definitions of the various categories of defect taxonomies (e.g. interface vs. non-interface), the definition of defect (especially with respect to injection time, detection time, environment, subjects, phase of data collection), etc.
- *A researcher needs to vary the classification to check the effects along the various values.* For example, in this instance we saw that to investigate the overall impact of interface-related defects, not only is it necessary to investigate the relative number of interface defects in the dataset, but the relative time required to fix those defects.
- There are insights to be gained from the collection and analysis of defects according to different classification schemes, independent of the scheme. Our results show that interesting abstractions can be drawn by comparing defect information opportunistically, based on points of similarity where they occur.

Based upon our experiences to date, we are evolving our methodology for building an effective set of folklore using empirical evidence from multiple studies. The methodology considers information found in papers published about the focus of the study. There is a specific approach to reading and extracting information from the paper. The information is extracted, summarized to create new knowledge that identifies possible context variables and expands the domain of study that is reported in an experiment in any one paper. This approach identifies three levels of abstraction: (1) the hypotheses from a particular study as presented in a paper and the information that can be extracted from that paper by identifying hypotheses, definitions, context variables, etc., (2) a broadened hypothesis from a family of focused related studies, built bottom-up by identifying the relevance of context variables to create integrated knowledge from two or more papers, and (3) vetted guidance or advice based upon empirical evidence abstracted so that it is useful to the software engineering community. Such abstractions are necessary for presenting information relevant for decision support to software developers and managers, for example through the Best Practices Clearinghouse project funded by the US Department of Defense [17] which aims at providing software developers and acquisition managers with robust knowledge based on empirical data.

References

1. Basili, V. R., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Sorumgaard, S.L., Zelkowitz, M.V.: The Empirical Investigation of Perspective-based Reading. Empirical Software Engineering, An International Journal, Volume 1, Number 2, pp 133-164, Kluwer Academic Publishers, October 1996.
2. Basili, V. R., Shull, F., Lanubile, F.: Building Knowledge through Families of Experiments, IEEE Transactions on Software Engineering, Vol. 25, No. 4, pp. 456-473, July 1999.
3. Brooks, A., Daly, J., Miller, J., Roper, M., Wood, M. (1996). Replication of experimental results in software engineering. Technical Report ISERN-96-10, Department of Computer Science, University of Strathclyde, Glasgow.
4. Lott, C. M., Rombach, H. D.: Repeatable software engineering experiments for comparing defect-detection techniques, Journal of Empirical Software Engineering, 1(3), 1996.
5. Wohlin, C., Runeson, P, Host, M., Ohlsson, M., Regnell, B., Wesslen, A.: Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers: Boston. 2000.
6. Juristo N., Moreno, A. M. (eds.): Lecture Notes on Empirical Software Engineering. World Scientific: New Jersey. 2003.
7. Mashiko, Y., Basili, V.R.: Using the GQM Paradigm to Investigate Influential Factors for Software Process Improvement, The Journal of Systems and Software, Volume 36, Number 1, pp 17-32, January 1997.
8. Chillarege, R., Bhandari, I., Chaar, J., Halliday, M., Moebus, D., Ray, B., Wong, M.: Orthogonal Defect Classification: A Concept for In-process Measurements, IEEE Transactions on Software Engineering, November 1992.
9. Basili, V.R., Weiss, D.M.: A Methodology for Collecting Valid Software Engineering Data, IEEE Transactions on Software Engineering, pp 728-738, November 1984.
10. IEEE. Software Engineering Standards. IEEE Computer Society Press, 1987.
11. [Endres75]

12. Weiss, D.: Evaluating Software Development By Error Analysis: The Data from the Architecture Research Facility, J. Systems and Software, V 1, 1979, 57-70.
13. Basili, V.R., Weiss, D.M.: Evaluation of the A-7 Requirements Document by Analysis of Change Date, Proceedings of the Fifth International Conference on Software Engineering, pp 314-323, March 1981.
14. Basili, V.R., Perricone, B.: Software Errors and Complexity: An Empirical Investigation, Communication of the ACM, vol. 27, no. 1, pp 42-52, January 1984.
15. Weiss, D.M., Basili, V.R.: Evaluating Software Development by Analysis of Changes: The Data from the Software Engineering Laboratory, IEEE Transactions on Software Engineering, pp 157-168. February 1985.
16. Selby, R.W., Basili, V.R.: Analyzing Error Prone System Structure, IEEE Transactions on Software Engineering, pp. 141-152, February 1991.
17. Dangle, K., Hickok, J., Turner, R., Dwinnell, L.: Introducing the Department of Defense Acquisition Best Practices Clearinghouse. CrossTalk, pp. 4-5, May 2005.