Using Defect Patterns to Uncover Opportunities for Improvement

Siddhartha Dalal[1], Michael Hamada[2], Paul Matthews, Gardner Patton

Bellcore
Morristown, New Jersey

1 Introduction

Although no one is happy to find defects in their software, defects are introduced and removed continually during software engineering processes, and it is practically necessary to acknowledge, record, and analyze those defects to make progress toward higher standards of quality. The Orthogonal Defect Classification (ODC) methodology, first introduced by Ram Chillarege and his colleagues at IBM [Chil92, Bhan94], provides one approach to analyzing defects. The ODC methodology has been adapted and extended for use within Bellcore, and a web based tool set, called the Efficient Defect Analyzer (EDA), has been developed to support ODC use on a large scale. This paper outlines how EDA has been successfully applied, potential pitfalls, and some possible future directions.

2 Context of ODC Use in Bellcore

Prior to the use of ODC, a common methodology for analyzing defects was for each project to form a defect analysis team, gather defect reports, and sit down together to review each defect one-at-a-time to reach consensus on the root cause and what could be done to prevent similar problems occurring. In some cases this process yielded some valuable insights, but it was also inefficient, requiring an average of several staff hours per defect to investigate the historical facts and reach agreement. To the extent that the defect analysis team assigned responsibility for each defect, conflicting interests limited objectivity and could make the process unpleasant. As a result, defect analyses were often limited to only the more serious defects reported by customers.

An ODC methodology was first tried as way to more objectively characterize and quantify defects so that priority could be given to addressing the larger problem areas. During these trial exercises, the defect analysis teams also discovered that they could do their work more quickly using ODC, form a coherent explanation of the facts, and make more convincing recommendations for software engineering process improvements. Such favorable experiences led to a corporate decisison to use ODC more widely, and to the creation of the Bellcore EDA tools to support a software systems organization of more than 2500 people.

Today, software defects are classified by the staff who detect and who resolve the defects as part of routine processes that record and track defects. These data are extracted from

[1] Authors in alphabetical order. The authors also want to acknowledge contributions to the work reported here by James Alberi, Yu-yun Ho and Kathy Musumeci.

[2] Currently at the University of Michigan.

the tracking systems, and loaded into a corporate repository with web based access for analysis.

3 Basics of Defect Pattern Analysis

A basic idea in ODC analyses is to classify each software defect using several orthogonal categories which describe different aspects of the defect. To make choices easier and to facilitate statistical analyses, the number of possible classification values for each category is intentionally small, about 10 or fewer. Although any one category captures a limited amount of information, when all the categories are considered together, a meaningful and rich characterization emerges.

The EDA tools support analyses of the relative frequencies of classification values, which we call "defect patterns". A general goal is to enable software product teams to improve engineering practices by discovering, investigating, explaining, and correcting unusual defect patterns.

3.1 EDA Categories

The following categories are used to classify each software defect, when the defect is detected,

- *Lifecycle* phase when the defect was detected
- *Trigger*, what was done to detect the defect
- *Impact* that the defect had or would have on the customer
- *Severity* of the impact from the customer's perspective

and when the fix is known,

- *Defect Type*, the type of software change that needed to be made
- *Defect Modifier*, missing or incorrect software
- *Defect Source*, change history of the code
- *Defect Domain*, generic platform and subsystem
- *Fault Origin* in requirements, design, or implementation.

Over time, the classification values used for each category have been tuned to be more meaningful to our EDA users, but have remained sufficiently general to make comparisons across products.

3.2 Episodes

For analyses, the classification data are segmented into distinct "episodes", usually defined by software product and release, and sometimes by subsystem or component. An episode should be a distinct work effort associated with a particular time period. We have found that comparing episodes is a key step in successful defect pattern analyses, because other episodes provide an objective basis for what patterns to expect, and concrete examples of what can be done differently.

3.3 Example Defect Pattern

The following table illustrates a defect pattern composed from two categories, Lifecycle and Defect Modifier.

N = 132		Lifecycle Phase			
		Code Inspection	Unit Test	Multiple Unit Test	Product Test
Defect Modifier	Missing	2%	3%	11%	20%
	Incorrect	16%	19%	17%	12%

These data could also be represented as a chart, for example by using a spreadsheet program or a chart widget. The number of defect records, 132, implies something about the statistical reliability of the pattern.

On inspection, the gray cells appear to be unusual pattern elements, being relatively higher or lower than other cells in the same row or column; one function of the EDA tools is to highlight highs and lows based on sound statistical and data mining principles, pointing out unusual features of the data that should be investigated and explained. In this example, relatively few Missing defects (e.g., code that was needed but not written) were found in the Code Inspection and Unit Test phases, while a lot were found in Product Test. One possible explanation is that programmers were intent on verifying the code that they had, but were less aware of the required functionality.

The one-way tables for Lifecycle Phase and Defect Modifier, below, suggest other possible pattern elements.

Lifecycle Phase			
Code Inspection	Unit Test	Multiple Unit Test	Product Test
18%	22%	28%	32%

Defect Modifier	Missing	36%
	Incorrect	64%

Relatively more defects were detected in later lifecycle phases, which is a danger sign but not very specific. And the split between Missing and Incorrect seems typical, until one examines the differences between lifecycle phases. In general, our experience suggests that insights are more likely to come from looking at two-way and higher classifications.

3.4 Seeing Patterns

In early ODC trials, an expert consultant helped defect analysis teams sort through the classification data by visually inspecting charts for patterns. But there were complications in extending this approach.

- Just 9 categories implies 36 two-way and 84 three-way combinations, and considering subsets of the data (e.g., "drilling down") further increases these numbers. Human observers are not usually effective in looking at data in so many different ways without mechanical assistance.
- Distinguishing meaningful patterns from random variation is not easy for the human eye, particularly when the number of software defects is relatively small. Chasing after noise can waste resources.
- Analysts did not always pick out the same patterns for special attention. The expert's judgment could not easily be replicated.

To scale up to wider use, the analysis methods were strengthened.

- More emphasis was placed on comparing patterns between episodes, that is, expectations were defined objectively using actual data. This led to the creation of a corporate repository, where classification data for different software products can be shared for analysis.
- Software tools were created to scan through all the category combinations and to highlight the more unusual patterns, using strong statistical filters to reduce noise. Analysts could then focus their attention on the highlights.

3.5 Whole Process

Finding unusual patterns is one step in a larger process, outlined below.

- Collect EDA data when a defect is first recorded and when the software is fixed. Compared with retrospective classification (e.g., at the end of a software product release), this saves time and ensures that the person doing the classification knows what the defect really was.
- Use a shared corporate repository to make comparisons with other episodes. A high (or low) percentage of one kind of defect may or may not be a problem, depending on what comparison data show; for example, a 36-64 split between Missing and Incorrect may not be alarming if those percentages are similar to other episodes, but a contrasting 64-36 split may be. Expect to make several comparisons, including with a previous release of the same software product, and with another contemporary product.
- Describe each unusual pattern element briefly in words. Collect these facts together, and write a summary in plain language that anyone can read. Spot check the original defect records to make sure that the summary is consistent with the details.
- Elaborate on the summary to tell a high level story of what happened that explains the data. Investigate to confirm the story, and to decide between alternative interpretations. Start by telling the story to a few key people who were involved in the episode.
- When the story seems solid, propose specific actions to address any issues raised by the story. Support and quantify the proposal with exhibits drawn from the data.

Keep in mind that these process steps may need to be executed quickly and at short intervals, perhaps as often as every week, to monitor software defects at critical stages and to take action to address problems that may be building up. Being able to see defect patterns in a few minutes makes this possible.

4 Case Study Example

Performing EDA analyses using a large repository of good quality data may seem easy, but that repository can take years to build up, and some people are discouraged from trying because they want results now to deal with today's pressing quality issues. The case study discussed below illustrates what can be done even with imperfect data.

4.1 Scenario

There were three software products in the case study scenario, named A, B and C. The focus was on product A, releases 2 and 3; the comparison episodes were product B release 3 and product C release 3. The defect analysis team was mainly concerned with field defects (i.e., defects discovered by their customers).

4.2 Comparing With a Previous Release

The first step was to compare the two releases of product A. There were 31 recorded field defects for release 2, and 50 for release 3.

The EDA analysis software did not highlight any of the classification patterns as different between the two releases. Since the defect analysis team members believed that the product components and software engineering processes were similar between the two releases, they decided to combine the data from the two releases to get greater statistical power for other comparisons.

4.3 Comparing With Another Product

Product A was then compared with a contemporary release of product B. However, the product B data as they stood contained only severity 2 (i.e., fix needed as soon as possible) defect records, so the analysis was restricted to the subset of severity 2 defects. There were 38 severity 2 field defect records for product B, and 10 for product A.

The following unusual pattern elements were highlighted by the EDA tools in comparing product A with B. (Click here to view all the one-way and two-way patterns.)

- *Impact*: Low percentage of Capability problems
- *Impact x Source*: High percentage of Reliability problems in Old Functionality
- *Trigger x Domain*: High percentage of Start/Restart problems in Server Processes

Next, product A was compared with product C, for severity 2 and then for severity 3 (i.e., fix less urgent) field defects. Product C had 134 severity 2 field defect records, and 175 severity 3 records; product A had 71 severity 3 records. (Product C was much larger than product A, with more software components, and was expected to have more total defects.)

The following unusual pattern elements were highlighted for severity 2 defects, comparing product A with C. (Click here to view all the one-way and two-way patterns.)

- *Trigger*: High percentage of Workload Stress problems
- *Defect Type x Trigger*: High percentage of Algorithm problems under Workload Stress
- *Defect Type x Impact*: High percentage of Algorithm problems have Reliability impact
- *Defect Source*: High percentage of problems found in Old Functionality
- *Defect Source x Impact*: High percentage of problems found in Old Functionality have Reliability impact

And for severity 3, the following unusual pattern elements were highlighted. (Click here to view all the one-way and two-way patterns.)

- *Domain*: High percentage of GUI problems
- *Defect Domain x Impact*: High percentage of GUI problems with Usability impact
- *Defect Domain x Fault Origin*: High percentage of GUI problems due to Implementation
- *Fault Origin*: High percentage of Requirements problems
- *Fault Origin x Defect Domain*: High percentage of Requirements problems for Server Processes
- *Defect Type x Trigger*: High percentage of Algorithm problems found under Workload Stress

So far, these are just lists of facts about the data that can be mechanically derived from looking at the highlighted cells in unusual defect patterns.

4.4 Comparing with Null Hypotheses

When no comparison episodes are specified, the EDA analysis software compares the null hypotheses that categories are independent and that classification values are equally frequent.

When the null hypothesis comparisons were done for the 10 product A severity 2 field defects, no pattern elements were highlighted (i.e., the null hypotheses could not be rejected). This is in contrast to the interesting story that emerged from comparing the same 10 records with the severity 2 field defects in projects B and C.

When all 81 product A records were compared with the null hypotheses, the statistical power was greater due to the larger number of records, and a bunch of pattern elements were highlighted. (Click here to view all the one-way and two-way patterns.)

- *Trigger*: High percentage found in Normal Mode of operation
- *Impact*: High percentages of Usability and Reliability problems
- *Severity*: High percentage of severity 3 problems

- *Defect Modifier*: High percentage of Incorrect
- *Defect Type*: High percentages of Checking, Assignment, and Algorithm
- *Defect Domain*: High percentage of GUI problems
- *Defect Source*: High percentage of Old Functionality, New Functionality, and Rewritten Code
- *Fault Origin*: High percentage of Implementation
- *Trigger x Defect Type*: High percentage of Algorithm problems found under Workload Stress

But many of these highs and lows had little meaning because some classification values tend to have relatively high or low frequency for just about every episode. For example, Normal Mode, Severity 3, and Incorrect are the usual majority values for their respective categories.

In practice, it is difficult to pick out salient features of the data without a realistic comparison

4.5 Putting the Story Together

The next step was write a coherent summary of the unusual pattern elements, as follows.

- "There were a relatively high number of reliability problems concentrated in older code."
- "Workload stress revealed a relatively high number of algorithm errors in older code that had severe reliability impacts."
- "A relatively high number of server problems were found during system restarts."
- "There were a relatively high number of GUI problems. However, these problems only affected usability and were not severe. A relatively high number of the GUI problems were due to implementation errors."

Based on followup investigation and discussion with other project members, project, the following story about project A emerged.

- "An optimization algorithm, originally programmed in release 1, failed in cases where high arrival rates caused a queue to grow beyond an expected limit. When a failure occurred, a database record could be stranded in an inconsistent state, causing a restart to fail."
- "New hire GUI programmers did not fully understand how to use the programming environment to implement the GUI design, and failed to handle some events properly. Also, as the design evolved, some of the online messages to the user got out of sync with the functionality."

4.6 Proposing Action

The defect analysis team proposed several changes to address what they perceived to be chronic problems in the way their project had been doing their work compared with other projects.

- "Provide more algorithm details in design documents, including assumptions about the runtime environment and possible limitations. In both design reviews and code inspections, reviewers should require information about the performance of algorithms under extreme conditions."
- "Exception handling code should be written to take over when expected limits are exceeded. Product testing should include high stress cases, and verify that exception handling code has been executed successfully."
- "Create a list of all information messages that can be sent to a user. Testing should include cases that invoke these messages."
- "Senior programmers should conduct monthly seminars on GUI design and programming; attendance should be mandatory for everyone involved in developing GUIs. A senior programmer with design knowledge should be involved in every GUI code inspection."

This list focuses on problems that were uncovered by the analysis. If no analysis methodology had been followed, the team would have risked making debatable proposals based on personal prejudices, rather than consensus proposals based on a shared perception of actual data.

4.7 Reviewing the Analysis

The analysis described in the case study was not ideal. For example,

- More defect records for earlier lifecycle phases would have helped fill in the picture of what kinds of defect were detected before product A was given to customers. For example, one might assume that GUI testing was slack if the customer found a lot of problems, or alternatively, that there so many GUI defects that even a very high removal rate was not good enough; without the data, we cannot say for sure.
- The Product B data should have included all defect records, not just the more severe.
- Some of the unusual pattern elements were ignored. Explicitly providing explanations, however obvious or trivial, would be better, because otherwise a significant fact sometimes gets overlooked.
- Taking more time to talk with their colleagues in the projects supporting Products B and C might have generated some additional ideas for improvements. Also, those projects would have benefited from a symmetric analysis comparing their products with product A.

Nevertheless, the story based on the defect patterns made sense to the team, fit the available data which helped convinced other people, and was the basis for specific action to improve software engineering practices across several lifecycle phases. And the defect analysis team's study began and ended on the same day, consuming less than 20% of the resources that would have been required for a traditional root cause analysis of the same defect records.

5 Potential Pitfalls

Our experience in rolling out the EDA methods and tools to a large organization suggests a number of potential pitfalls.

- *Already knowing the answers.* Although it seems incredible, the stories told about software defects sometimes bear no clear relationship to any data, but instead reflect longstanding prejudices. Requiring defect analysis teams to explain the actual data goes a long way to inspire fresh thinking. Early involvement of expert consultants often seems necessary to induce people to pay more attention to the data.
- *Incomplete data.* Software defects that are not classified, or not recorded at all, can introduce statistical bias. One solution is to establish clear criteria for what defects ought to be recorded, and to require EDA classification before they can be cleared. For example, a general rule might be to record all defects, found in any lifecycle phase, that could affect a customer, and to require complete EDA classifications before changed software can be delivered for inclusion in a release.
- *Improper or inconsistent classification.* Strange classifications tend to stand out as unusual patterns, and are easily detected. Our experience suggests that most people can learn to classify defects fairly well with an hour or two of practice in a group training seminar. Ongoing peer review, for example, in the context of defect analysis teams, and monitoring of data quality by corporate metrics experts are recommended.
- *Pounding down nails.* A common first impulse in defect analysis teams is to identify classification values with larger percentages of defects, and to do something about reducing those numbers. This may make sense in some cases, for example, when zero defects are expected, but it does not make sense in general, because proper patterns are unlikely to have uniform cell frequencies. Also, in some lifecycle phases, low (not high) percentages may be more cause for concern if suspected defects are not being detected. Teams need to think about reducing the total number of product defects, and also about achieving a proper pattern for each lifecycle phase and defect detection activity.
- *Isolation.* Due to the way software defect statistics have been handled in the past, some projects may be reluctant to share their defects data, and skeptical of the data that other projects provide, preferring to analyze their own data in isolation. But comparisons with other episodes are a key source of learning, and additional outside perspectives help gel constructive plans for improvements. As suggested by the case study, it is not easy to learn from one's own experience alone.

6 Future Directions

Our EDA experience has been consistently positive and continues to expand. As with other metrics and data analysis efforts, success depends on assiduous attention to data quality and the patience to build up data to establish baselines and do comparisons.

Enabling defect analysts to do their work efficiently by providing appropriate infrastructure and analysis software has been a central focus of our EDA tools effort.

Going forward, we expect to evolve the EDA tools and methods in several directions.

- Software defects detected later in a product release cycle seem more compelling due to the nearness of customer impact. Detailed records of field faults and defects uncovered by organized testing are commonly available, and provide a natural starting point for EDA analyses. However, we also believe that some of the more troublesome defects are introduced and could be detected much earlier. We are currently working to extend our EDA practices to requirements and design reviews and other early processes that constrain what the software later becomes.
- Although reasoning based on statistics can be quick and effective, we think that many analysts would benefit from advanced visualization techniques. For example, we have used emerging web 3D programming techniques to show categorical data as multifaceted objects that can be viewed from different angles and taken apart. Combined with statistical highlighting, we think such techniques will enable users to understand more EDA analyses at a glance.
- We think that automation of EDA analyses can reach the point where the human analyst need only confirm or choose between conjectures offered by the analysis software. Not all the information needed to complete a software defects analysis is available in databases: it will still be necessary for the analyst to do some reality checking and provide feedback. This level of artificial intelligence would enable more analysts to be successful with less effort and training.

If you have been thinking about trying approaches like those discussed in this paper, we encourage you to do so.

7 References

[Chil92] R. Chillarege et al., "Orthogonal Defect Classification -- A Concept for In-Process Measurement", *IEEE Transactions on Software Engineering*, Vol. 18, No. 1, 1992.

[Bhan94] I. Bhandari et al., "In-process improvement through defect data interpretation", *IBM Systems Journal*, Vol. 33, No. 1, 1994.

Paul Matthews

Paul Matthews is a software engineer in Bellcore's Applied Research area, located in Morristown, New Jersey. Mr. Matthews has 21 years experience in major telecommunications research and development companies, including assignments as a statistics and data analysis consultant, software system designer and developer, software engineering course developer and instructor, advanced technology prototyping and technology transfer agent, and his current assignment as a research scientist supporting quality assurance. Mr. Matthews has a Ph.D. in experimental psychology from Stanford University, where he did research in cognitive sciences and statistics. He is keenly interested in objective assessments of software engineering practices, and in project management and process improvement based on data.