Submission of accepted article 111768 to 'IEEE Software'

Accelerating Learning from Experience: Avoiding Defects Faster

Lutz Prechelt (prechelt@computer.org)
Fakultät für Informatik
Universität Karlsruhe
76128 Karlsruhe, Germany
current address: abaXX Technology AG, Forststr. 7,
70174 Stuttgart, Germany
+49/711/61416-1500, Fax: -1111

April 25, 2001

Abstract

Over time, any programmer will learn from repeated mistakes. Can we speed up this learning and save many unnecessary repetitions of similar errors? This article presents a technique called "defect logging and defect data analysis" (DLDA) for doing exactly this. The technique is inspired by Humphrey's Personal Software Process but has much lower learning cost. It was validated in a controlled experiment which showed significantly faster improvement in the experiment group compared to the control group, despite a small group size (which makes it hard to obtain statistical significance). We urge practitioners to try the technique, which produces noticeable improvements at low cost and with very low risk.

Keywords: error, defect, process improvement, quality management, controlled experiment

Introduction

All programmers learn from experience. A few of us are rather fast at it and learn to avoid a particular kind of mistake after they have made it only once or twice. Others are very slow and make a similar mistake hundreds of times. Most are somewhere in between: They reliably learn from their mistakes, but it is a slow and tedious process. The probability of making a structurally similar mistake again decreases only slowly during possibly some dozen repetitions. That way, it often takes years before one has properly learned a certain rule — positive or negative — for one's own behavior.

The topic of this article is how to accelerate this process of learning from mistakes for an individual programmer, no matter whether learning is currently fast, slow, or very slow.

One successful approach to improving individual learning is the Personal Software Process (PSP) developed by Watts Humphrey [1]. A report on the success obtained in the PSP course was published in this magazine [2]. The techniques applied by the PSP for improving learning from mistakes are based on defect logs. Each and every defect found during design, design inspection, coding, code inspection, testing, or operation is recorded along with a description, defect type classification, origin and cause of the error leading to the defect (if known), and the time required for locating and repairing the defect. When enough defect data has been collected after a few months, the data is analyzed for recurring types of mistakes and the insights gained are converted into entries on inspection checklists and changes to the development process itself. Besides this defect prevention technique, the PSP provides effort estimation and time planning techniques and a complete framework for process definition, process measurement and control, and continuous process improvement.

However, the PSP approach has a drawback. The effort for learning the methodology is immense. In the standard form (aimed at undergraduate or graduate software engineering courses), 15 complete working days (spread over 15 weeks) are required. Many project managers are interested in the PSP at first — until they hear about the learning effort. Even a reduced PSP variant such as the one presented by Humphrey in [3] and aimed at introductory programming education is considered by most industrialists to be far too expensive to learn. Furthermore, the volume of bookkeeping proposed by the PSP is so large that data quality may become dubious [4].

The problem is that you cannot learn the PSP just by listening to a few presentations and then applying the techniques in your daily work. Under normal work pressure, very few programmers are able to keep up the discipline for following all those cumbersome PSP techniques before they have experienced the advantages to be gained from them. However, the advantages can only be experienced if one follows the techniques at least for a while. That is why the course is required: for providing a pressure-free playground on which to learn about the effectiveness of the PSP techniques.

The present article reports how it is possible to learn and apply defect logging and defect data analysis (DLDA) in isolation, without any PSP course at all. The effort is only about one half day and the requirements for self-discipline are moderate. The technique applies not only to coding, but rather to most phases and activities in the software process. A controlled experiment indicates that even the first application of the technique by a programmer results in a significant improvement. This result suggests that it is possible to obtain an important part of the PSP benefits without the substantial investment into a complete PSP course.

The DLDA technique

To clarify the terminology, if a programmer commits a mistake, we call this an *error*, one possible result of an error is a *defect* in the software document. Once turned into code, the execution of a defect can result in a *failure* of the software. Thus, errors (by human beings) and failures (by machines) are events, but defects are structural deficiencies. Much of the software process is concerned either with avoiding or with detecting and removing such deficiencies. A explicit focus on avoidance and early detection is a characteristic of a mature process. Improving such activities is the aim of defect logging and defect data analysis (DLDA), but it can be applied even in otherwise immature software processes.

As the name suggests, DLDA consists of two separate phases. Defect logging is performed during all software construction activities where defects in the product might be found, that is during requirements definition, requirements review, design, design review, implementation, code review, testing, and maintenance. Defect data analysis is a process improvement activity that is performed only rarely, when enough defect data has been collected after some weeks or months.

```
1999-04-13 13:09:37 bts
1999-04-13 13:12:23 be
1999-04-13 13:22:24 ee cd we ty # multiplied instead of adding
1999-04-13 14:39:34 be
1999-04-13 14:46:35 ee cd ma om # forgot to reset stdDeviation after first calc
```

Figure 1: *Two example defect log entries. The first defect was detected (*`be`*, "begin error") at 13:12 hours, three minutes after the start of the testing phase (*`bts`*, "begin test"), and was located and removed (*`ee`*, "end error") ten minutes later at 13:22. It had been introduced in the coding phase (*`cd`*), is of structural type "wrong expression" (*`we`*) and the reason for making it was a typo (*`ty`*). The second defect was of type "missing assignment" (*`ma`*) and was produced due to omission (*`om`*, i.e., in principle the programmer knew that it had to be done). A typical DLDA describes defects by between 2 to 10 phases, 10 to 50 defect types, and 7 defect reasons (omission, ignorance, commission, typo, missing education, missing information, external).*

Defect logging consists of creating a protocol entry whenever the presence of a new defect in a software document is detected. The entry is started by recording a time stamp. When the defect has been localized, understood(!), and repaired, the entry is completed by another time stamp and additional descriptive information about the defect, for instance the exact location of the defect, the type of the defect according to some fixed defect type taxonomy, the phase when the defect was presumably created, the estimated reason why it was created (such as: lack of information, defects elsewhere, trivial omission, trivial mistake, etc.), and possibly a short or long verbal description; see Figure 1 for an example. After some practice and when using a compact format, recording this information is much simpler and faster than it sounds, provided it is done with a tool that records the time stamps and performs some simple consistency checking.

In the defect data analysis phase, the programmer clusters the defects found into groups of related ones according to whatever criteria seem appropriate. The grouping is guided by the defect categorizations used in the defect log. These groups are then analyzed to understand the most frequent and the most costly types of mistakes and the programmer tries to understand why s/he makes these mistakes (root cause analysis). Again, there is no prescribed method for doing this. The analysis is entirely data-driven and relates to whatever understanding a programmer has of his/her own software process. Tabulations of defect data by work phase, defect type, repair cost etc. can be created automatically and are used to aid the analysis.

Unfortunately, defect logging requires quite a bit of discipline and most programmers are unable to keep that up in the middle of their normal work. That is why *coaching* is the third important ingredient of DLDA. When a programmer starts with DLDA s/he should first be instructed how the technique works and must be motivated by at least one convincing example of an insight gained by DLDA — ideally an insight of somebody s/he knows well. Then during the first few days of defect logging, a coach must be around who stops by from time to time, asks for the most recent defects, reminds the programmer of logging, and discusses any how-to questions that might have come up. Coaching will only work with programmers who *want to* apply DLDA — there is no point in trying to force the technique on anybody not willing to give it a chance. Obviously, the coach should be somebody who does not only know how to do DLDA, but also is really convinced of its benefits. Once enough defect data has been collected (about one hundred defects are a good start), the coach should also counsel during defect data analysis. The coach should explain how to summarize the data and may also help with the actual conclusions insofar as the programmer does not find them her/himself.

The experiment

We validated DLDA in a controlled experiment. We prepared two programming tasks, both of an algorithmic nature. Task 1 consisted of computing the “skyline” for a set of “buildings” represented as rectangles described by 2-d coordinates. Task 2 is computing the convex hull for a set of 2-d points, also described by their coordinates. In both cases, the algorithms to be used were given in coarse pseudocode (4 statements for task 1, 7 statements for task 2). Both tasks involve computations with 2-d coordinates. The resulting programs had 80 to 206 lines of code for task 1 and 93 to 160 for task 2.

Each participant of the experiment worked alone and solved both tasks, starting with task 1. There were two groups: the experiment group, which used DLDA, and the control group, which did not. The members of both groups were asked to protocol the time required for each work phase (design, coding, compilation, test) and to track and subtract the time for interruptions. Only the experiment group was additionally asked to perform defect logging as described above. The total time required by the participants who finished the tasks ranged from 3.6 to 10.7 hours for task 1 and from 1.8 to 6.0 hours for task 2.

The experiment group received a one page instruction sheet for defect logging before they started task 1. After reading it, their understanding of the method was checked by a one page scenario for which they were to fill in the defect log entries. Mistakes made in that test, if any, were then explained by the coach; the experimenter acted as the coach for all participants. The defect data analysis was part of the task for the experiment group. A defect data recording template was given to the participants as well as a two page template for the defect data analysis plus one page of concrete instructions for its use. In the experiment, we used a scaled-down variant of DLDA. Besides a short description, the repair time, inject phase, and remove phase of the defect, only the error reason class was recorded, but no defect type class. Consequently, defect data analysis was also simplified accordingly. On the other hand, since the tasks are small, we asked the participants to log even simple defects such as syntactical defects and missing declarations.

During the experiment, the participants were unintrusively observed. If a participant failed to fill in the time log or the defect log correctly, the coach reminded him; if necessary he clarified again how to apply the method. The coach painstakingly watched out to help only with the DLDA technique, but not at all with the programming task itself. A finished program was accepted only when it passed a certain fixed set of tests.

The participants were 18 male graduate or senior undergraduate students of Computer Science, Electrical Engineering, and Computer Engineering from the University of Massachusetts in Dartmouth. Several of these dropped out during the experiment: 3 were too inexperienced and were not capable of solving the first task, 5 were unwilling to invest the time for the second task after they had finished the first. This leaves 10 participants, 5 in each group. Fortunately it was possible to pair these 10 participants into 5 pairs of subjects with similar levels of experience, thus ensuring a reasonable balance between the groups. The members of each pair were assigned to the groups randomly. 7 of these 10 had one or several years of experience as professional programmers.

For all participants, the source code of each program version compiled was saved along with a time stamp. From this data we later determined the number of defects introduced into a program and the time required for eliminating each of them.

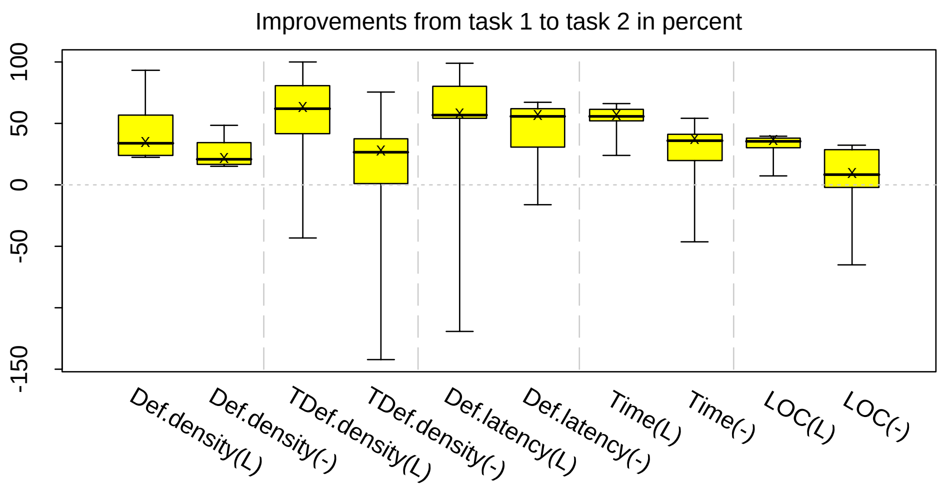

Figure 2: *Performance changes of the defect logging group (L) versus the non-logging group (–) for several metrics: defect density, test defect density, defect latency (average defect lifetime), total work time, program length. In each boxplot, the whiskers indicate the value of the best and worst participant, the box those of the second best and second worst, and the middle line those of the median.*

Results

Given only 5 participants in each group one might expect that obtaining meaningful results is impossible. But as we will see, this is not the case.

From the data collected during the experiment, we computed the following basic metrics for each task of each participant. The total time required for the task, the resulting program size in lines of code (LOC), the number of compilations, the number of defects inserted in any program version, and the number of defects removed during testing. From these, we then compute the defect density, i.e., the number of defects inserted divided by the size of the program. The test defect density accordingly considers the defects removed during test only. The defect latency expresses the average lifetime of a defect from its insertion to its removal as visible in the compiled program versions.

Due to our small group sizes it would be dangerous to assume the exact equivalence of the two groups and to compare their performance on one task directly. Since the two tasks are obviously different from one another, it is also not meaningful to compare the first task to the second task within one group. Hence, the most sensible approach to evaluating our data is to compare the performance *changes* from first to second task of one group to the performance changes of the other group. Therefore we compute the improvements (in percent) for the above metrics by comparing task 1 results to task 2 results for each group. The most interesting results are shown in Figure 2.

In that plot, each box and the median line within it indicate the improvements of the middle three subjects of one group for one metric. The whiskers indicate the other two subjects. We see that with at most one exception per group there was always an improvement from the first task to the second. Only subject 1 (from the logging group) and subject 8 (from the control group) performed worse in their second task.

The improvement differences between the experiment group and the control group are not huge, but appear quite substantial. Are they real or accidental? For answering this question we can use a statistical hypothesis test (see text box). A one-sided Wilcoxon Rank Sum Test informs us that the reductions in median defect density and test defect density each have a probability of only 11% of being accidental — reasonable evidence that DLDA improves defect prevention. This is corroborated by the corresponding comparison of the mean reductions, using a bootstrap-based test, which indicates a 9% and 10% probability for defect density and test defect density, respectively, that the observed differences are random, non-systematic events. For the defect latency, the same tests indicate a 35% and 54% chance, respectively, that there is no real improvement, which suggests that DLDA did not cause faster defect *removal*, at least in this experiment.

The chance that the median work time reduction is accidental is only 5% (for the mean: 3%). With a probability of 0.8, the improvement percentage is larger by at least 16 in the experiment group, although we should be aware that part of this improvement is because the defect logging is getting faster itself — in the first task, the DLDA participants were undoubtedly slowed down a bit by their unfamiliarity with the technique. However, on average they logged only 25 defects in 6 hours, which is a minor effort in any case.

The difference in program length is quite interesting. During the defect data analysis, several participants recognized that they should have spent more effort on properly designing their program. The more careful design in task 2 then resulted in more compact programs as well. The median difference to the control group is accidental with a probability of only 5% (for the mean: 2%).

We see that despite the small group sizes we can be reasonably sure of the evidence found in the experiment. DLDA results in better defect prevention and thus increases productivity.

One might fear that these results do not transfer to professional programmers at all. It would be plausible that our inexperienced subjects had so much more room for improvement than an experienced software engineer that DLDA might be worthless in practice despite the experiment results. However, we found some evidence to the contrary in our data. For the test defect density, our data shows a clear trend that the more experienced among our subjects actually obtained a *larger* improvement than the others. This is true no matter whether experience is measured in years of professional programming experience as well as measured by the length of the largest program ever written by the subject.

Statistical hypothesis tests

A statistical hypothesis test is a mathematical procedure for comparing two data samples (sets of related values). For instance, the Wilcoxon Rank Sum tests compares the medians of two samples, the t-Test compares the means, a Bootstrap-based means-differences test compares the means but does not require assuming a normal distribution, etc. Based on the variation within the observed data, such a procedure will compute the probability (called the p-value) that the apparent differences between the two samples are not real but rather merely due to chance. For example in our case, if that probability is small, we will be willing to believe that the observed additional improvements in the DLDA group are *not* accidental and will consider the experiment good evidence of the usefulness of DLDA. Comparing the medians indicates whether for a single individual we should expect an improvement or not. In contrast, comparing the means indicates whether we should expect an improvement when averaging across a team (a group of individuals).

Conclusion

Defect logging and defect data analysis (DLDA) appears to be a viable technique for accelerating learing from experience: programmers learn to prevent mistakes faster than usual. For most programmers, learning DLDA will require a coach experienced in the technique, but apart from that the learning cost is low. In a controlled experiment we found that a group using DLDA for only one small programming task (a half day) solved a second task faster and with smaller defect density than a control group — despite the fact that the experiment used only a simplified version of DLDA due to time constraints.

Nevertheless, for someone who is sceptical about the usefulness of DLDA, our experiment will not be completely convincing, because its participants were not experienced as professional software engineers. However, we believe in the validity of the DLDA approach and hence we urge practitioners to try it for themselves with a few colleagues. It is a low-cost, low-risk technique with considerable potential benefits; a champion who quickly takes on the role of the coach can usually be found.

Further Research

The research presented above leaves a number of questions open. Selecting participants: Our experience with teaching the PSP indicate that more than half of all programmers appear to be unable to keep up the self-control required for a process such as defect logging. This appears to be a personality issue. How can we quickly and safely determine whether training somebody in DLDA will lead to actual usage later? Just asking the candidate solves (only) about half of the problem.

Tools: What tools provide best support for defect logging? The web page http://wwwipd.ira.uka.de/PSP/ provides some tools for defect logging, a tool for defect data summarization, and a defect classification standard.

Detailed methodology: What defect classification categories are most useful? In which contexts? What criteria are best for defect analysis? How far can defect analysis be standardized and simplified before losing value? Our only insights so far are that for most (but not all) people it appears that more value comes directly from the logging, rather than from the analysis. However, a useful defect classification scheme may play an important role even then.

Coaching: What minimum intervention by the coach is sufficient in general? How can the coach best adapt the interventions to the needs of the specific trainee? Should DLDA best be combined with pair programming?

Acknowledgements

The experiment was carried out by Georg Grütter. We thank Michael Philippsen for commenting on a draft of this article and our experimental subjects for their participation.

References

[1] Watts S. Humphrey. *A Discipline for Software Engineering.* SEI series in Software Engineering. Addison-Wesley, Reading, MA, 1995.

[2] Watts S. Humphrey. Using a defined and measured personal software process. *IEEE Software*, 13(3):77–88, May 1996.

[3] Watts S. Humphrey. *Introduction to the Personal Software Process.* SEI series in Software Engineering. Addison-Wesley, Reading, MA, 1997.

[4] Philip M. Johnson and Anne M. Disney. The personal software process: A cautionary case study. *IEEE Software*, 15(6):85–88, November 1998.