# USING DEFECT ANALYSIS FEEDBACK FOR IMPROVING QUALITY AND PRODUCTIVITY IN ITERATIVE SOFTWARE DEVELOPMENT

Pankaj Jalote
Department of Computer Science and Engineering
Indian Institute of Technology Kanpur
Kanpur – 208016; India

Naresh Agrawal
Infosys Technologies Limited
Electronics City, Hosur Road
Bangalore – 561 229; India

## *Abstract*

In today's business where speed is of essence, an iterative development approach that allows the functionality to be delivered in parts has become a necessity and an effective way to manage risks. Iterative development allows feedback from an iteration to influence decisions in future iterations, thereby making software development more responsive to changing user and business needs. In this paper we discuss the role of defect analysis as a feedback mechanism to improve the quality and productivity in an iteratively developed software project. We discuss how analysis of defects found in one iteration can provide feedback for defect prevention in later iterations, leading to quality and productivity improvement. We give an example of its use and benefits on a commercial project.

## *Introduction*

A project using the traditional waterfall model of developing software assumes that that requirements are stable, and delivers the entire software at one shot in the end. Unchanging requirements, unfortunately, do not exist in reality, and the "big bang" approach of delivery entails heavy risks, as the users do not know till the very end what they are getting. To alleviate these two key limitations, an iterative development model can be employed. In an iterative development, software is built and delivered to the customer in iterations – each iteration delivering a working software system that is generally an increment to the previous delivery. Iterative enhancement [1]

and spiral [6] are two well-known process models that support iterative development. More recently, RUP[15], agile methods [10], and XP [4] also promote iterative development. The timeboxing process model for shortening delivery times is also an iterative approach [19, 20].

The commonly used iterative development approach is organized as a sequence of iterations, with each of the iterations delivering parts of the functionality. With this type of iterative development, as working software is released in stages, feedback from one release is available for next iterations. Quick feedback is one of the key aspects of XP [4].

Most often the feedback taken from an iteration is on the functionality and user aspects of the software. However, iterative development also renders itself to feedback about the process, which opens the possibility of process improvement in later iterations of the project. That is, it can allow for software process improvement within a project itself. Note that this approach is distinct from process improvement across the organization – something that is the focus of frameworks like the CMM [13], and ISO9000 [21].

In this paper we focus on how analysis of defect data from an iteration can provide valuable feedback for preventing defects in later iterations of the project. The premise of defect prevention is that there are some causes behind the defects that get injected. If the causes can be understood, then efforts can be made to eliminate them, thereby reducing the number of defects getting injected. In other words, by analyzing the defect data to find the root causes for the defects getting injected, and then developing solutions that attack the root causes found, the level of defect injection can be reduced.

Defect prevention can improve both quality and productivity. If the number of defects injected reduces, then the quality improves as the number of residual defects in the delivered software reduces. Furthermore, if we inject fewer defects, fewer defects need to be removed, leading to a reduction in the effort required to remove defects, thereby increasing productivity. Defect prevention has been used by organizations with good benefits [18]. DP is a Key Process Area at the level 5 of the CMM, hence all high-maturity organizations employ it.

The common approach for employing defect prevention is to have an organization-wide program where experience from different projects can be employed for other projects. Leveraging the learning from past defects to avoid future defects is somewhat harder across projects with different characteristics, and also requires a much wider program to support it. On the other hand, learning from defects from within a project and then leveraging the lessons in the project itself is likely to offer more focused and effective solutions. Such an approach can be deployed quickly as it is applied within the limited context of a project and is hence easier to implement. Projects employing iterative development offer a platform as each iteration is complete in itself and can be analyzed as a project.

In this paper we describe an approach to exploit this potential. In this approach, at the end of each iteration, the defect data is compiled and then analyzed for identifying root causes. Based on this analysis, defect prevention actions are proposed, which are then employed in the project for future iterations. We give a detailed example of how this approach has been implemented on a commercial project at Infosys Technologies Ltd, a large software house headquartered in Bangalore, and describe the results.

## *Measurements for Defect Analysis*

In some sense the goal of all methodologies and guidelines is to prevent defects. For example, a design methodology gives a set of guidelines that if used will give a good design. In other words, the design methodology aims to prevent the designer from introducing design defects by guiding him along a path that produces good and correct designs.

However, by defect prevention (DP) we mean learning from actual defect data from a project with the goal of developing specific plans to prevent defects from occurring in the future. As the main goal of DP is reduction in defect injection and consequent reduction in rework effort, it is best if suitable measurements are made such that impact of DP can be quantitatively evaluated. That is, a project employing DP should be able to see the impact of DP in the injection rate and on the rework effort on the project. For both of these proper metrics have to be collected. Furthermore, suitable data needs to be collected to facilitate the root cause analysis for DP.

The measurements needed for evaluating the effectiveness are defects and effort. For defects, data on all the defects found and their types is needed. This data is easily available if projects follow the practice of defect logging, as is the case in most mature organizations. To facilitate defect analysis, for each defect, its categorization in a fixed set of categories should also be recorded. A classification like the one proposed by the IEEE standards [23], or by the orthogonal-defect classification scheme [22] can be used. Frequently, organizations log information like detection stage, injection stage, etc to facilitate different types of analyses. Details about the different parameters recorded during defect logging are given in [9].

For understanding the impact of DP on rework, the effort spent on the project needs to be recorded with suitable granularity such that rework effort can be determined. Specifically, for each quality control activity, the rework effort should not be clubbed together with the activity effort but must be recorded separately. Effort logging generally requires that each member of the project team record the effort spent on different tasks in the project in some effort monitoring system. Frequently, different codes are used for different categories of tasks and for most of the major tasks the effort is divided into three separate categories – activity, review, and rework. With this type of

categorization, rework effort for each phase can be determined. Details about the system and codes used for effort reporting are given in [9].

These measurements about defects and effort are sufficient to do defect analysis and prevention, as well as quantify the impact of DP. Note that DP can be done, and its impact on the defect injection rate can be determined, even if the effort data is not available. However, without the effort data, the impact of DP on rework cannot be determined.

### *Deploying Defect Analysis and Prevention*

In a project, to use DP, all the DP related activities should be planned, like any other major task. We propose that the project planning stage be augmented by the following tasks to support defect prevention activities:

- Identify defect prevention team within the project
- Have a kick-off meeting and identify existing solutions
- Set defect prevention goals for the project
- Get the DP team trained on DP and causal analysis, if needed

Most of the activities relating to DP planning are self-explanatory. A project identifies a team that will do perform the DP analysis (obviously, the actual solutions that are to be executed to prevent defects will have to be performed by everyone in the project.) A kick-off meeting is held in the start to raise the awareness and identify the solutions that may be available somewhere in the organization. The training for DP team on DP and causal analysis is done, if needed.

DP is a strategy to achieve higher quality and productivity in future iterations by leveraging the experience of an earlier iteration. As with many tasks, setting suitable goals helps in focusing the effort and monitoring the progress. In a project, the DP goal can be in terms of reduction in the defect injection rate in later iterations. For example, projects in Infosys frequently aim to achieve 20% to 30% reduction in the injection rate with the feedback from the first iteration. Note, however, that setting a DP goal is not essential for performing DP – it, however, helps set a target and evaluate the performance against it. The steps for performing defect analysis and prevention in an iterative project are:

- At end of an iteration, collate defects data
- Identify most common types of defects by doing Pareto analysis
- Perform causal analysis and prioritize root causes
- Identify and develop solutions for root causes
- Implement solutions
- Review the status and benefits of DP at end of next iteration

Collating defect data is a simple task if a suitable defect tracking tool is used. The next step for defect prevention is to draw a Pareto chart from the defect data. Pareto analysis is a common statistical technique used for analyzing causes, and is one of the seven primary tools for quality management. In this step, the number of defects found of different types is computed from the defect data and is plotted as a bar chart in the decreasing order. Frequently, with the bar chart, a line chart is also plotted on the same graph showing the cumulative number of defects as we move from types of defects given on the left of the x-axis to the right of the x-axis. The Pareto chart makes it immediately clear in visual as well as quantitative terms which are the main types of defects, and also which types of defects together form 80-85% of the total defects.

The Pareto chart helps identify the main types of defects that have been found in the project so far. For reducing these defects in future, we have to find the main causes for these defects and then try to eliminate these causes. Cause-effect (CE) diagram is a technique that can be used to determine the causes of the observed effects [16]. The main purpose of the CE diagram is to graphically represent the relationship between an effect and the various causes that can cause that effect to occur. The understanding of the causes helps identify solutions to eliminate them.

The building of a cause effect diagram starts with identifying an effect whose causes we wish to understand. For defect prevention, an effect is of the form "too many errors of type X". To identify the causes, first some major categories of causes are established. In manufacturing, for example, these major causes often are manpower, machines, methods, materials, measurement, and environment. For software, the standard set of major causes defined for causal analysis of defects can be process, people, technology, as these are the main factors that impact the quality and productivity [9, 13]. With the effect and major causes, the main structure of the diagram is made – effect as a box on the right connected by a straight horizontal line, and an angular line for each major cause connecting to the main line.

For analyzing the causes, the key is to continuously ask the question "why does this cause produce this effect?" For example, for a project with too many GUI defects the questions are of the type "why do people cause too many GUI defects" or "why do processes cause too many GUI defects." This is done for each of the major causes. The answers to these questions become the sub-causes and are represented as short horizontal lines joining the line for a major cause in the CE-diagram. Then the same question is asked for the causes identified. This "Why-Why-Why" process is repeated till all the root causes have been identified, i.e. we have reached the causes for which asking a "Why" does not make sense. When all the causes are identified and marked in the diagram, the final picture looks like a fish-bone structure and hence the cause-effect diagram is also called the fish-bone diagram, or Ishikawa diagram after the name of its inventor.

Once this diagram is finished, we have identified all the causes for the effect under study. However, most likely the initial fishbone diagram will have too many causes. Clearly, some of the causes have a larger impact than others. Hence, before completing the root cause analysis, the top few causes are identified. This is done largely through discussion. For defect prevention, this whole exercise can be done for the top one or two categories in the Pareto analysis.

Once the root causes are known, then the next step is to think of what can be done to attack the root causes, such that their manifestation in form of defects reduces. That is, think of preventive actions for the causes. The basic paradigm is the age old adage "prevention is better than cure". Some common prevention actions are building/improving checklists, training programs, reviews, use of some specific tool. Sometimes, of course, drastic actions like changing the process or the technology might also be taken.

To see the impact of DP, and to exploit further opportunities that may exist, we suggest that DP exercise be done after each iteration, for the first few iterations. Once the results indicate that that benefits are tapering off, the DP activities may be stopped.

The preventive solutions are action items which someone has to perform. Hence, the implementation of the solutions is the key. Unless the solutions are implemented, they are of no use at all. At Infosys, along with the solution, the person responsible for implementing the solution is also specified. These action items are then added to the detailed schedule of tasks for the project and their implementation is then tracked like other tasks.

An important part of implementing these solutions is to see if it is having the desired effect – namely, reducing the injection of defects and thereby reducing the rework effort expended in removing the defects. Further analysis of defects found after the solutions have been implemented can give some insight into this question. The next analysis for defect prevention done at the end of the next iteration can be used for this purpose. Besides tracking the impact, such follow-up analysis has tremendous reinforcing value – seeing the benefits convinces people like nothing else. Hence, besides implementation, the impact of implementation should also be analyzed.

### *An Example*

Let us illustrate the whole process through an example of a commercial project executed at Infosys. This project used a variation of RUP [11] for execution and had three construction iterations (using the RUP terminology.) Summary of the analysis of defect data after the first construction iteration is shown in Table 1 – this is a simple type-wise breakup of defects. The Pareto chart for this defect data is shown in Figure 1.

Table 1: Summary of defect data for the first iteration

| Defect Type | Logic | Standards | Redundant code | User Interface | Architecture |
|---|---|---|---|---|---|
| No of Defects | 19 | 17 | 11 | 8 | 2 |

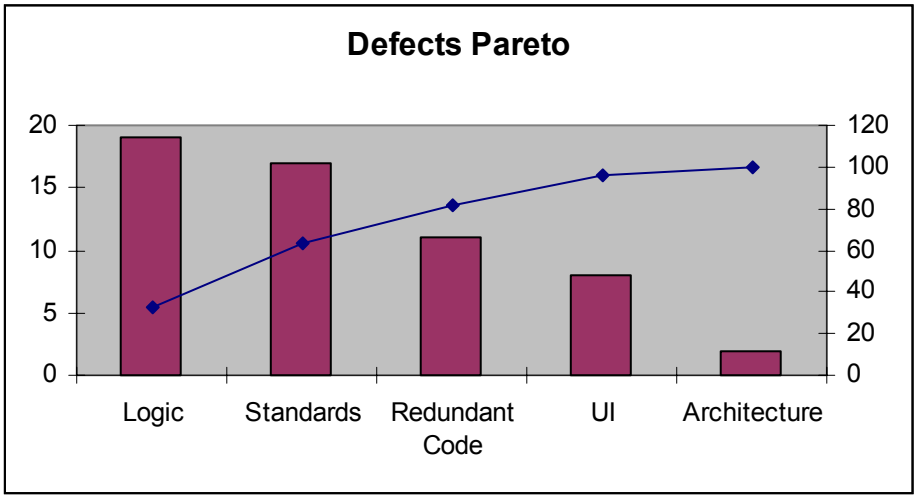


Figure 1: Pareto Chart for Defects

In the first iteration we know that at least 57 defects were injected. The effort for that phase is also known from the effort data for the project. From these two, we get the defect injection rate for just the build phase as 0.33 defects per hour. The project goal was to achieve a reduction in the defect injection rate to about half this level.

In the defect analysis meeting, it was decided that to reduce the defect injection rate significantly, all the top three categories of defects – logic, standards, and redundant code – will be tackled. A brainstorming session was held to identify the root causes and the preventive actions that are possible. A standard brainstorming procedure was followed – first all the possible causes that anyone suggested were listed, then the ones that were identified as the main culprits were separated out. For these causes, possible preventive actions were discussed and finally agreed. (At Infosys, the brainstorming meetings for the root cause analysis and for determining the solutions are done in the same session, and the final result is generally reported in a tabular form, even if a fish-bone diagram is drawn during the brainstorming session.) The final result

of the causal analysis meeting was a table giving the main root causes and the preventive actions that have to be implemented. Part of this table is shown in Table 2. Note that most of these preventive actions are schedulable activities and hence were scheduled in the project schedule and then later executed and monitored like other project tasks.

Table 2: Root causes and preventive actions for ACIC project

| **Defect Type (number of defects)** | **Root Cause** | **Preventive Action** |
|---|---|---|
| **Standards (17)** | Oversight | Do a walkthrough of the standards with developers. |
| | Coding Standards not updated | Update coding standards for imports, naming, and user interface |
| **Redundant Code (11)** | Lack of understanding of object model and database | i) Training on Database structure ii) Conduct a short training on the object model. |
| | lack of understanding of existing code | Arrange code reading sessions. |
| **Logic (19)** | lack of understanding of existing code | Arrange code reading sessions. |
| | Lack of understanding of database and object mode. | Same as earlier. |
| | Lack of understanding of use cases | Do a requirement walkthrough. |

The preventive actions were implemented for the next iteration. Whether these measures are reducing the defect injection rate or not can only be checked through the measurement data in future iterations in which these DP solutions have been implemented. In this example which had three iterations, the defect injection rate after the next two iterations was also determined. The result of the analysis done after the other two iterations is shown in Figure 2.

This chart clearly shows the impact of implementing the preventive actions on the defect injection rates in the second and third iterations – the defect injection rates fell from over 0.33 defects per person-hour to less than 0.1! The reduction in injection rate from 2$^{nd}$ to 3$^{rd}$ iteration was small, suggesting that feedback through defect analysis from early iterations is more valuable and scope for improvement reduces after one set of actions has been implemented. In other words, it suggests that detailed root-cause analysis and determination of preventive actions perhaps need to be done only for early iterations – for future iterations just keeping track of the injection rate might suffice.

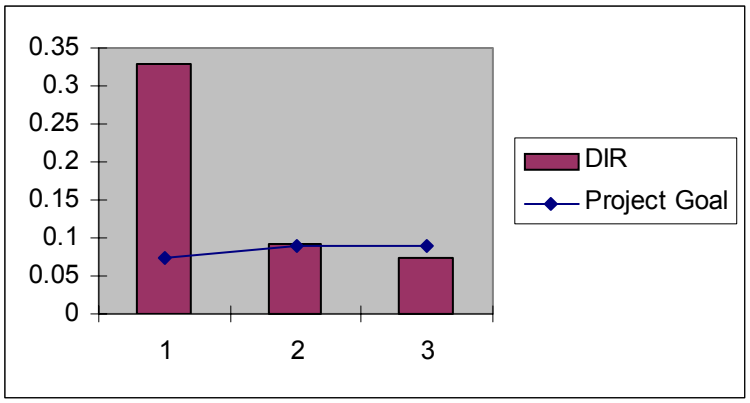

Figure 2: Defect Injection Rate in Different Iterations

Reduction in defect injection implies that there are fewer defects to be detected and fixed. Consequently, there should be a reduction in the rework effort in later iterations. The rework effort as a percent of overall iteration-development cost is shown in Figure 3. As we can see, the pattern of reduction in rework effort is similar to the pattern of reduction in the defect injection rate – the rework effort reduced from about 15% to less than 5% in the second iteration and reduced further to about 3% in the third iteration. This rework effort is obtained from the effort and the rework effort data. This clearly illustrates the power of defect analysis feedback from one iteration into future iterations.

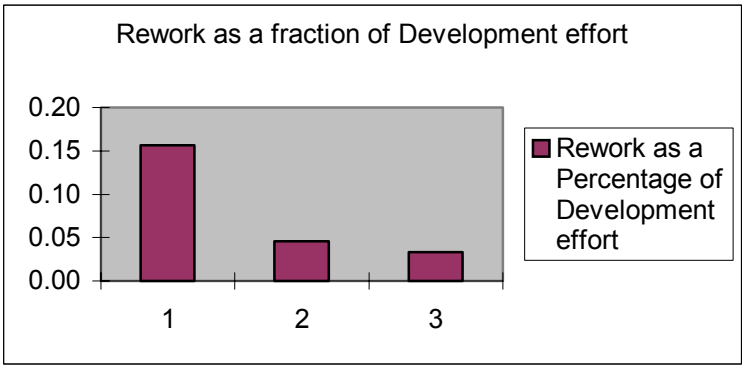

Figure 3: Rework reduction in ACIC due to defect prevention

## *Summary*

Iterative development models are now the preferred approaches for developing software as such models avoid some of the problems of the waterfall model and are better suited for the fast changing world of today. In an iterative development, the software is developed in a series of iterations,

each iteration delivering a working system which is then used to build the next version of the system with more capabilities. Besides delivering working systems quickly, in iterative development the feedback from the software developed in one iteration becomes an invaluable input for development in the next iterations.

The iterative development renders itself to the possibility of leveraging the experience of one iteration for improving the development process of next iterations. In general, software process improvement tries to leverage the experience of an organization for the benefit of the projects. Such an improvement model uses experience from past projects for future projects. Though very useful, frequently experience from one set of projects is not directly usable in a new project. On the other hand, experience from one iteration of a project should be directly useful and relevant to the other iterations, leading to the possibility of improving a project's process by in-process data and experience.

In this paper we propose the use of defect data from one iteration for defect prevention in future iterations. The defect data is collected during the development. At the end of an iteration, the defect data is analyzed using a structured analysis process leading to identification of root causes of defects and suggestions for attacking the root causes to prevent the occurrence of such defects in future. These suggestions are the process improvement possibilities actions for future iterations.

We have applied the technique to many projects and have shown the application in detail on one project. This example clearly illustrates how defect injection rate falls rapidly through the use of this technique. The rework effort as a percentage of total effort also falls significantly, leading to an improvement in productivity also. Experience with other projects strengthens the hypothesis that structured feedback from one iteration can be very effective in improving quality and productivity in future iterations.

This concept of using a project's data for improving the process of the project itself can also be applied to other process models, though not as naturally as in iterative development. In a waterfall type model, some checkpoints will have to be established at which analysis is done and results fed back. In long running projects, such an analysis can be done at regular intervals. At Infosys we have seen benefits of applying this technique in these projects as well.

## *References*


1. V. R. Basili and A. Turner, Iterative enhancement, a practical technique for software development, *IEEE Transactions on Software Engg.,* 1(4), Dec 1975.

2. V. R. Basili, Ed., *Tutorial on Models and Metrics for Software Management and Engineering,* IEEE Press, 1980.

3. V. R. Basili and H. D. Rombach, The experience factory*, The Encyclopedia of Software Engineering*, John-Wiley and Sons, 1994.

4. K. Beck, *Extreme Programming Explained*, Addison Wesley, 2000.

5. E. J. Chikofsky, Changing your endgame strategy, *IEEE Software*, Nov. 1990, pp. 87, 112.

6. Cockburn, *Agile Software Development*, Addison Wesley, 2001.

7. Collier, T. DeMarco, and P. Fearey, *A* defined process for project postmortem review, *IEEE Software*, pp. 65-72, July 96.

8. J. L. Hennessy and D. A. Patterson, *Computer Organization and Design,* Second Edition, Morgan Kaufmann Publishers, Inc., 1998.

9. P. Jalote, *CMM in Practice – Processes for Executing Software Projects at Infosys*, SEI Series on Software Engineering, Addison Wesley, 2000.

10. C. Jones, Strategies for managing requirements creep, *IEEE Computer,* 29 (7): 92-94.

11. P. Kruchten, *The Rational Unified Process – An Introduction*, Addison Wesley, 2000.

12. W. W. Royce, Managing the development of large software systems, *IEEE Wescon,* Aug. 1970, reprinted in *Proc. 9th Int. Conf. on Software Engineering (ICSE-9)*, 1987, IEEE/ACM, pp. 328-338.

13. Software Engineering Institute, *The Capability Maturity Model for Software: Guidelines for Improving the Software Process*, Addison Wesley, 1995.

14. C. Larman, *Applying UML and Patterns,* 2nd Edition, Pearson Education, 2002.

15. C. Larman and V. R. Basili, "Iterative and Incremental Development: A Brief History", June 2003, IEEE Computer.

16. D. N. Card, "Learning from our mistakes with defect causal analysis", *IEEE Software,* Jan-Feb 1998.

17. D. N. Card, "Defect causal analysis drives down error rates", *IEEE Software,* July 1993.

18. R. Mays et al., "Experiences with defect prevention", *IBM Systems Journal,* 29:1, 1990.

19. P. Jalote et. al., "Timeboxing: A process model for iterative software development", Journal of Systems and Software, 2004, 70:117-127.

20. P. Jalote et. al., "The Timeboxing process model for iterative software development", in Advances in Computers, 2004, Vol 6, pp 67-103.

21. International Standards Organization, ISO900-1, Quality Systems – Model for Quality Assurance in Design/Development, Production, Installation, and Services, 1987.

22. R. Chillarege et. al. Orthogonal defect classification – a concept for in-process measurements. *IEEE Transactions on Software Engineering,* 18(11):943:956, Nov 1992.

23. IEEE, Std. 1044-1993. IEEE standard classification for software anomalies, IEEE.