Six Sigma Software

While most software quality efforts focus on requirements, design, code and test, this method focuses on tuning delivered software. Yes, it would be better to prevent the kind of problems we see in software, but applications continue to be written by people using requirements and designs that can be flawed. Software is rarely released, it escapes.

IT managers and application users often expect a new software project or enhancement release of an application to be flawless, and then are stunned by the additional staffing required to stem the tide of rejected transactions.

The secret is to:
1. Quantify the cost of correcting these rejected transactions
2. Understand the pareto pattern of rejected transactions
3. Analyze a sample (30) of the rejected transactions one-by-one to determine the root cause, revise the requirements, and modify the system to prevent the problem.

Quantify the Costs

The first step in what I call the "Dirty Thirty" process is to identify the number of rejected transactions and the associated costs. In working with one telecom company, we found a 17 percent level of rejected service orders (170,000 parts per million):

There were over 30,000 errors per month which, at an average cost of $12.50 to fix (correction group wage cost only), cost $375,000 per month. Over 50 temporary workers had been hired to deal with the backlog of unfixed errors. The objective was to cut this level of rejects in half by the end of the year.
Understand the Pareto Pattern

All systems have routines to accept, modify, or reject incoming transaction data. These are assigned error codes and dumped into error buckets to await correction. In the service order system, the application handled much of the modification, but it still left significant quantities of defects to be corrected manually:

There were over 200 different error codes, but only six (3%) accounted for over 80% of the total rejected transactions. Two directly affected service; four affected the customer records:

It only took about three days to gather the data and isolate these transactions as the key ones to focus on.

Analyzing the Dirty Thirty

The next step was to convene root cause teams to investigate 30 rejects of each error type. It took a week or more to get the right people in the room to investigate each type of error. The right people included the IT systems analyst, error correction, and service order entry. To attempt to do all six at one time with the same people would have been foolish. The errors required different subject matter experts and the root causes were too different. By restricting ourselves to just one error type per team, we were able to find the root causes in just one half day meeting per team.

In IBM’s original 360 operating system, 4% of the code caused 64% of the errors. This pareto pattern holds true in all software.

4% of transactions cause over 50% of the errors.

In 2000 Service Affecting Fallout

Jan 2000 Service Affecting Fallout

Jan 2000 Record Affecting Fallout
To prepare for the meeting we printed out at least 100 examples of each error (to ensure we had enough). Then,

1. Using all of the online systems, we investigated the root cause of each rejected transaction. Again, we restricted ourselves to analyzing just one transaction at a time.

2. As the team agreed on the cause of the rejected transaction, I kept a stroke tally for each root cause. Gradually, as we looked at more and more transactions, a pattern would reveal itself. Sometimes it only took 25 transactions; sometimes it took 50, but a pattern would reveal itself clustered around one or more root causes. The great thing about evaluating transactions one at a time is that you verify your root causes as you go.

3. Once the team had identified the root causes, we would stop analyzing and spend an hour defining the new requirements. Most of the time, the requirements were too tight, sometimes too loose, and occasionally non-existent. The systems analyst would then convey these to the programming staff for implementation.

Analyzing Results

It took four months to implement the revisions, but it was worth it. By midyear, the changes completely eliminated the top two service-affecting errors, and three of the four record-affecting changes. It cut total errors from 31,121 down to 23,954 per month—a 77 percent reduction in total errors. This reduction translated to saving over $250,000/month—$3 million per year.

Common Problems

The core elements of any application involve searching for and then adding, changing, or deleting data. Most applications
assume a perfect world, where the data is only created or modified by the system. This is rarely the case. Most systems have a wealth of backdoors used to fix faulty data quickly. Upstream and downstream systems have their own backdoors to fix faulty transactions, so perfect data continues to be a mythological assumption that fosters faulty requirements and designs.

The requirements for adding, changing, and deleting data are often: 1) too loose, 2) too tight, or 3) nonexistent which leads to errors and rejected transactions that must be corrected manually by people hunched over computer terminals for eight hours a day.

**Conclusion**

Although it would be better if we could prevent errors in requirements, design, code and test, every system release could benefit from a simple, yet rigorous approach to analyzing and eliminating post-implementation errors. The "Dirty Thirty" process is ideal because the data required to implement it is collected by most systems automatically. Then all it takes is four-to-eight hours of analysis to identify the root cause of the error. Most of the time, the root cause will reside in the requirements.

One of the positive by-products of this approach is that systems analysts learn first hand how their requirements and designs most often fail. This allows them to learn how to make their next set of requirements or designs more robust.

It also gives the user a closer look at the intricacies of software and the complexities involved.

Until software engineering finds ways to prevent all of the possible defects inherent in software development, the Dirty Thirty process will provide a simple way to tune up a system after release to move it ever closer to Six Sigma performance (3.4 defects per million transactions).

Pick one of your mission-critical applications that suffers from too many bugs. Find the 4-50. Analyze the root causes. Change the system and reap the rewards.

**Jay Arthur**, the KnowWare® Man, works with companies that want to systematically cut costs and boost profits. He is the author of Six Sigma Simplified and the QI Macros SPC software for Excel. He is a B.S. in Systems Engineering and the author of five John Wiley and Sons books on software engineering. He can be reached at: 888-468-1537, knowwareman@mindspring.com

Cut your costs and boost your profits.

Use the Dirty Thirty process to tune up systems after release. Build it into your process.