# Charting the Progress of System Development Using Defect Data

**Rex Black;** President and Principal Consultant,
RBCS, San Antonio, TX

**Key Words:** Test management, development management, project management, bugs, defects, bug reports, metrics, development project, opened/closed, found/fixed, defect removal, closure period, root cause, subsystem.

# Abstract

Aggregate defect data (bug reports), presented graphically, allow testers to give development, project, and executive management "dashboard" information they need to drive development projects to successful conclusions. I discuss four charts that distill meaning from test findings in terms of product stability, quality, bug repair, root cause, and affected subsystems.

# Introduction

While bug reports are useful by themselves, you can discover some interesting facts by analyzing them in aggregate. Defect data form the basis of various quality control charts in a number of industries, and computer software and hardware are no exception. In this paper, I will introduce you to four simple yet powerful, sophisticated defect analysis charts. The charts provide information on project, process, and product quality. Using bug data, you can generate defect analysis charts that show patterns in the process of defect removal, the root causes of bugs, the effectiveness of bug management, and the parts of the product that cause the most problems.

These charts are also excellent for communicating with management for three reasons. First, test metrics, graphically presented, highlight and illustrate test results that are difficult to explain using the raw data, which usually consist of a thick stack of bug reports. Second, getting people to focus for an hour-long bug review is hard, but presenting and explaining four charts fits into the available attention span. Third, the perspective of these charts encourages the viewers to manage the project and process according to key indicators, rather than to resolve the crisis du jour.

# Case Study

In the following paper, I use a case study approach to introduce these charts. The case study is a hypothetical software development project to implement a word processor called SpeedyWriter. I assume that the test team will execute a three-phased test effort as outlined in Table 1. The case study is presented from the perspective of a person looking back on the project on September 5, 1999, which gives us the benefit of hindsight. However, these charts are at their most useful *during* a project; with practice, you'll be able to understand the trends when you see them developing. I've assigned pertinent data

points somewhat randomly but to illustrate specific circumstances. The charts are representative, though cleaner and clearer than those from a real project.

| Phase | Cycle | Start Date | End Date | Bugs Found |
|---|---|---|---|---|
| Component test | 1 | 7/19/99 | 7/25/99 | 25 |
| | 2 | 7/26/99 | 8/1/99 | 20 |
| | 3 | 8/2/99 | 8/8/99 | 5 |
| Integration test | 1 | 8/2/99 | 8/8/99 | 20 |
| | 2 | 8/9/99 | 8/15/99 | 15 |
| | 3 | 8/16/99 | 8/22/99 | 5 |
| System test | 1 | 8/16/99 | 8/22/99 | 10 |
| | 2 | 8/23/99 | 8/29/99 | 5 |
| | 3 | 8/30/99 | 9/5/99 | 0 |
| First customer ship | — | 9/13/99 | — | — |

*Table 1: The Bug Location and Test Schedule for SpeedyWriter*

The SpeedyWriter team uses a basic bug tracking database. To prepare the defect analysis charts discussed in this paper, the following fields are necessary:

- Report opened date
- Report closed date
- Root cause
- Affected subsystem

To prepare these charts, you can use any bug tracking system capable of collecting these four data points for each bug report, and exporting at least these four data fields into a file format readable by a spreadsheet program. (I used Microsoft ® Excel to create the charts in this paper.) Of course, a useful bug tracking database must capture and report a lot more information than these four fields, but that's a topic for another conference.

In the following sections, I use bug reports-documented symptoms of problems observed by testers-as a numerical proxy for actual bugs-defects present in software or hardware. Because of duplicate bug reports, bug reports that turn out to be the result of test system or tester failure, and other noise in the bug tracking database, this is a liberal approximation. On five projects consisting of both hardware and software, I have observed defect overestimation by bug report counts between 26 and 32 percent, with 27 percent the most frequent value. Therefore, the approximation is defensible, and, assuming your bug reporting database captures root cause data, you can correct your estimates dynamically.

# Opened/Closed

The first and most important chart shows daily and cumulative bug opened and bug closed counts. It illustrates interesting trends in both defect location and repair, which often correspond to important events in the project timeline. It also shows the stability of the product, and the quality gap. Figure 1 provides an example of an opened/closed chart.

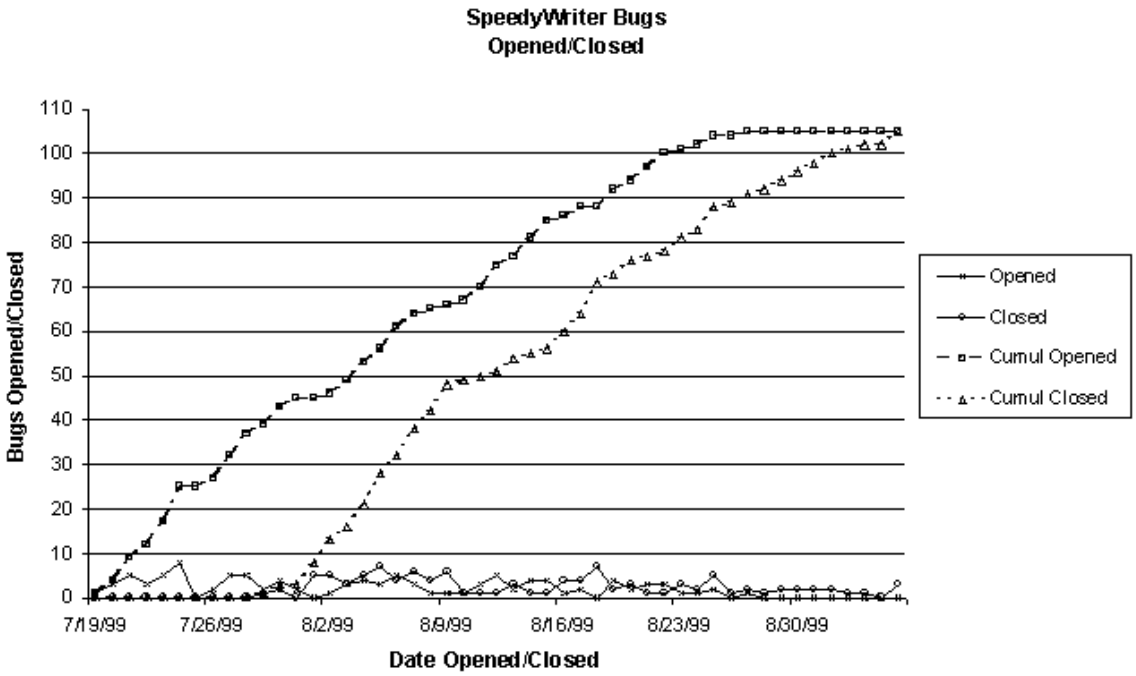



*Figure 1: Ideal SpeedyWriter opened/closed chart*

This information-rich chart provides answers to a number of questions. First, are you ready to ship the product? Since the number of bugs in a given program is unknown but is essentially a fixed number once development ends, test projects eventually hit a point at which further testing produces diminishing returns. When the cumulative opened curve—the top curve on the chart in Figure 1—levels off at an asymptotic limit, testing is usually considered complete, at least for the phase currently under way. (The asymptote actually indicates the fading of the test system's ability to find bugs in the current phase. Given a good test system, the bugs found represent the bugs most likely to torment the customer, and the fading of the test system is consistent with customer indifference.) Conversely, a cumulative opened curve that refuses to flatten indicates that you have plenty of problems left to find. On the chart in Figure 1, the limit was hit around August 23: the second cycle of system testing revealed few additional bugs, and then none were found in the final round of system testing.

Next, have you finished fixing bugs? Once development winds down, developers usually start to catch up with the problems. At about the same time, the cumulative opened curve starts to flatten. Consequently, the cumulative closed curve—the lower curve on the chart in Figure 1—begins to converge with the cumulative opened curve. Given the latency period that arises from the release management process, the closed date for a given bug does lag the developer fix date. If you want to add this, you will need a field to in your bug tracking database that tracks when bugs are fixed.

At a more general level, is the bug management process working? It is working well in this example: the closed curve follows right behind the opened curve, indicating that the project team is moving the bugs quickly toward resolution.

Finally, how do milestones in the project relate to inflection points, changes in the slope of the cumulative opened or cumulative closed curves? The overlapping of the phases in

the example obscures this relationship somewhat, but often when you move from one test phase to the next, you will see a spike in the cumulative opened curve. Such rises are gentle in our idealized example, but these transitions can be dramatic and even downright scary on some projects. As developers move from writing new code or engineering new hardware to fixing bugs, you should see an upward turn in the cumulative closed curve. A "bug scrub" meeting, where the technical and management leaders of a project gather to decide the fate of all known bug reports, can result in a discontinuous jump in the cumulative closed curve.

Such a chart is easily prepared using the `COUNTIF()` function in Excel. Create a worksheet that counts, for each date during the test phases, how many bug reports were opened and closed. Then accumulate these numbers day-by-day. Finally, use the Excel chart wizard to create the chart.

To explore the use of opened/closed charts, let's look at three troublesome scenarios that represent those unpleasant projects in which all test managers eventually participate. All of these examples assume the SpeedyWriter testing schedule outlined in Table 1. For the first, imagine that during the system test phase the bug find rate remains high and refuses to level off. The opened/closed chart that results appears in Figure 2. Notice the deceptive leveling in the second cycle of system test (8/23/99 through 8/29/99), where the opened curve appears to flatten, only to leap up sharply in the third cycle (8/30/99 through 9/5/99). If the project team ships the product on September 13 as scheduled, they can expect many failures in the field.

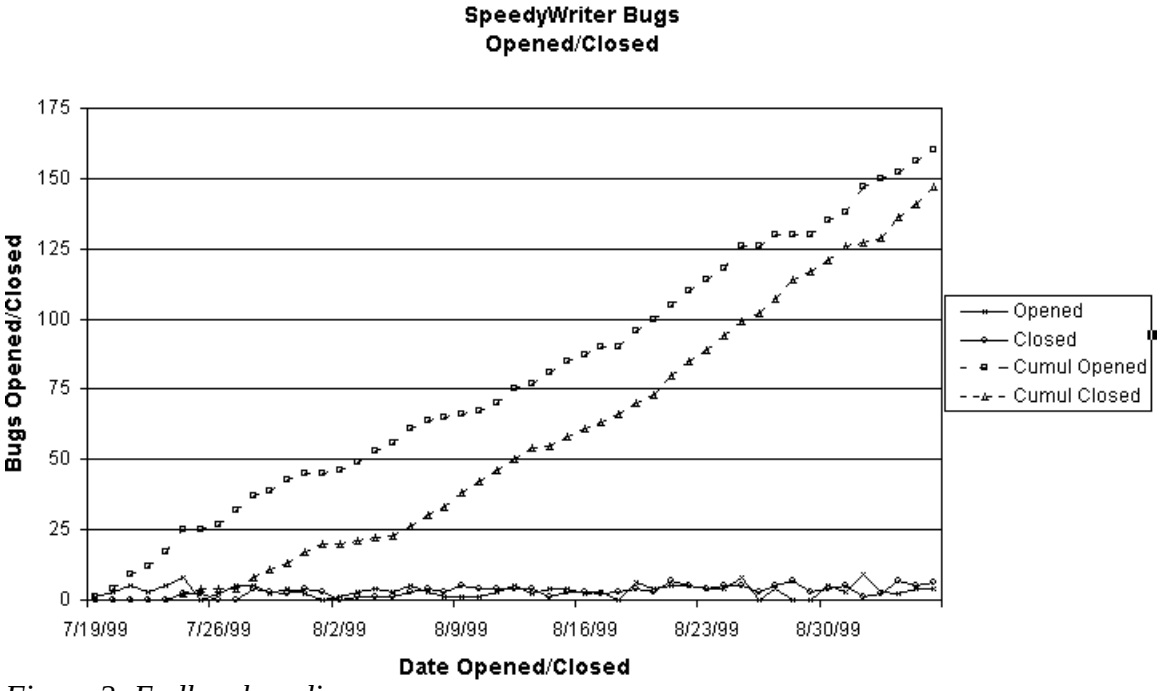



*Figure 2: Endless bug discovery*

For the second scenario, let's assume that development is ignoring some of the bugs. The developers have convinced themselves that about 25 of the bugs that test has reported are beneath contempt and can be disregarded. Figure 3 shows the opened/closed chart for this scenario. Until about August 20, the chart in Figure 1 (the idealized example) and the chart in Figure 3 are not radically different: on that date, the quality gap is about 20 in the

former chart, whereas it is about 30 in the latter. Ten bugs out of 100 one way or the other three weeks before the first customer ship is not a catastrophe. But as the second and third cycles of system testing unfold (8/23/99 through 8/29/99, and 8/30/99 through 9/5/99, respectively), it becomes clear that the gap is not narrowing. Unless you bring the pernicious bugs to project management's attention around August 23, the development team will deliver a *fait accompli*. Overturning their decision to ignore these bugs even a week later will require a slip in the delivery date.

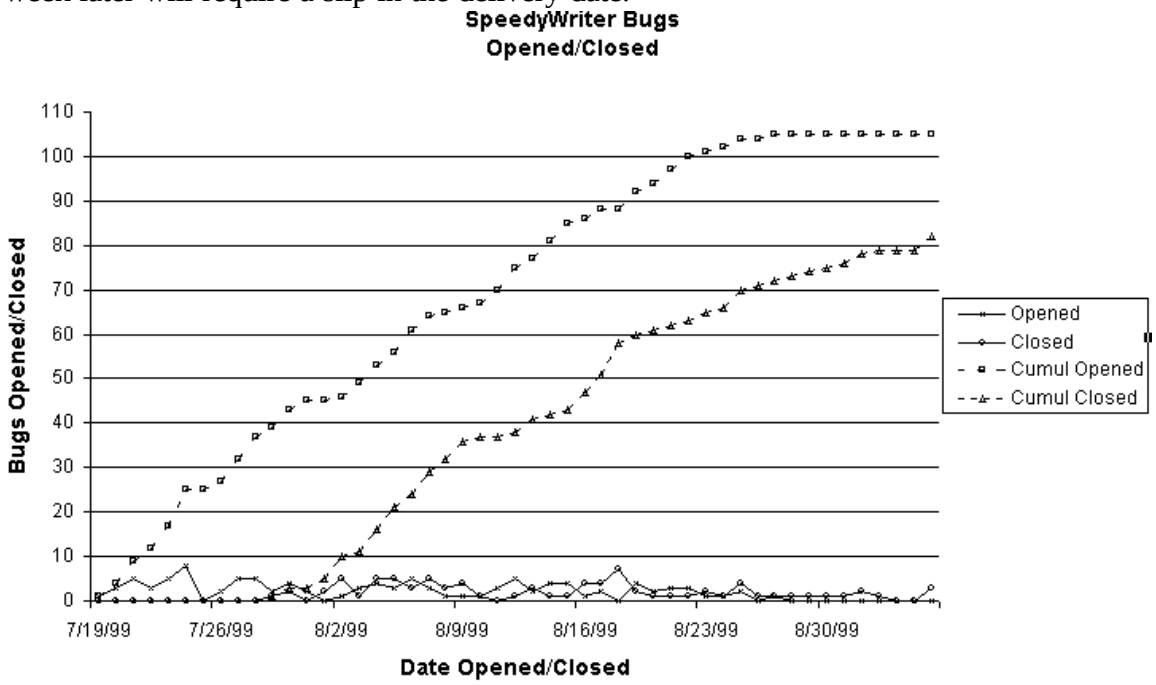



*Figure 3: Ignored bug reports*

Finally, suppose that the developers and the testers are both doing the right things at a technical level, but the bug management process isn't working. The development manager doesn't notify you when bugs are fixed and ready for confirmation testing, and you don't follow up with your testers to make sure they close bugs that pass confirmation testing. Also, testers don't bother to report bugs when they find them but instead wait until Thursday or Friday each week. Then they enter their findings, some dimly remembered, into the bug tracking database. Figure 4 shows how the opened/closed chart looks in this example. The trouble here is that you can't tell whether a jump in a curve represents a significant event or simply indicates that some people are getting around to doing what they should have done earlier. If you use this chart as part of your project dashboard, your gauge is jumpy.

# Closure Period

While we're on the topic of bug report management, let's proceed next to the closure period chart, which shows the daily and rolling (project to date) closure period for bug reports. Closure period (a.k.a. closure gap) is complicated to calculate, but it has a simple intuitive meaning: the closure period measures development's responsiveness to test's bug reports. Daily closure period refers to the average number of days between the opening of a bug report and its resolution for all bug reports closed on the same day.

Rolling closure period is the average for all closed bugs, including the current day and all previous days. Figure 5 shows the closure period chart for the SpeedyWriter project. As you can see, the daily plot tends to "pull" the rolling plot toward it, although the ever-increasing inertia of the rolling average makes it harder to influence as the project proceeds.

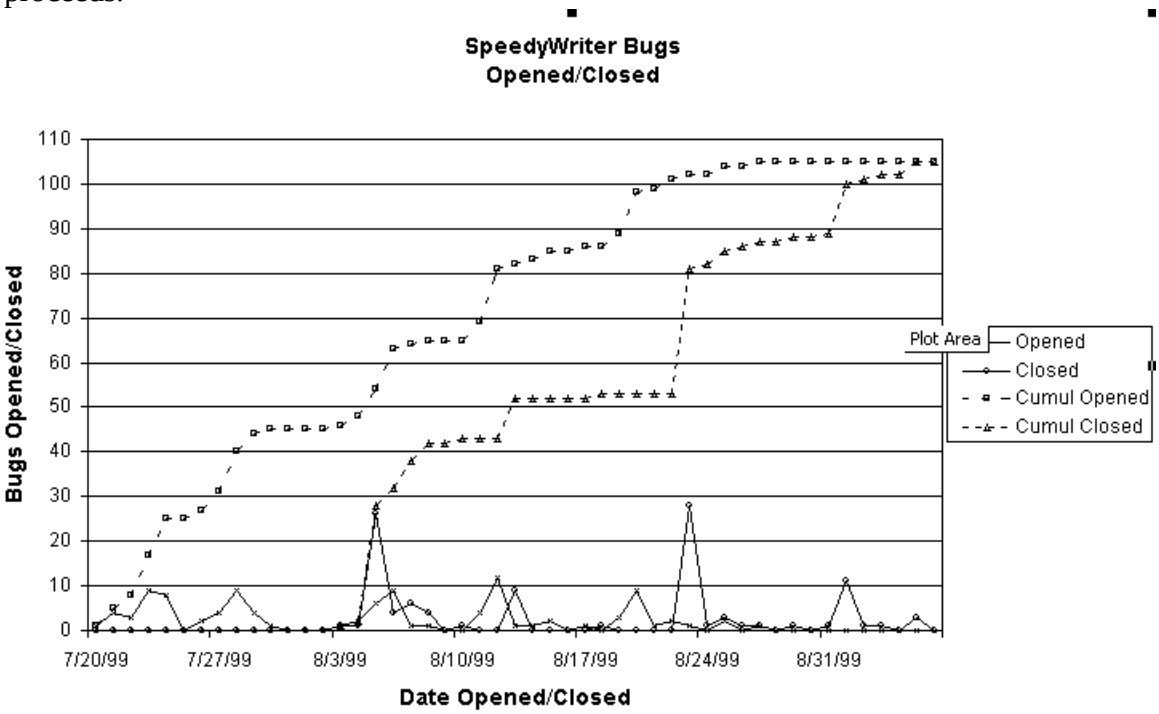



*Figure 4: Poor report management*

It's useful to look at closure period in terms of stability and acceptability. A stable closure period chart shows a relatively low variance from one day to another, with the slope of the rolling closure curve remaining almost constant and close to 0. In addition, the daily closure period fluctuates randomly around the rolling closure curve, staying within a few days in either direction.

On an acceptable closure period chart, both the daily and rolling closure period curves fall within the upper and lower limits set in the project or test plan for bug turnaround time. Although the pressures of the typical project make it hard to believe, there is indeed a lower limit for an acceptable closure period. Bugs deferred the day they are opened pull the daily closure curve toward 0, but the bug remains in the product. Furthermore, an acceptable daily closure curve does not exhibit a significant trend toward either boundary.

A closure period chart that is both stable and acceptable indicates a well-understood, smoothly functioning bug management process. The ideal is a low number with a downward or level trend since an efficient bug management process drives bugs through their state transitions to closure with all deliberate speed. The closure period in Figure 5 is stable and, if management is realistic in its expectations, acceptable. Bugs tend to get fixed in about a week and a half, which is a good pace if you assume one-week test cycles and formal release management.

Unlike opened/closed charts, there are not archetypal "bad" closure period charts that indicate particular types of failures in the development process. If the daily closure period shows too much variability, then that might indicate sporadic bug fixing, but it could also mean that a mix of simple functionality and complicated stability bugs are in the bug fix queue. An upward trend in the rolling closure period can indicate a slowing of the fix process, but it may also imply that development is focusing on the difficult bugs.

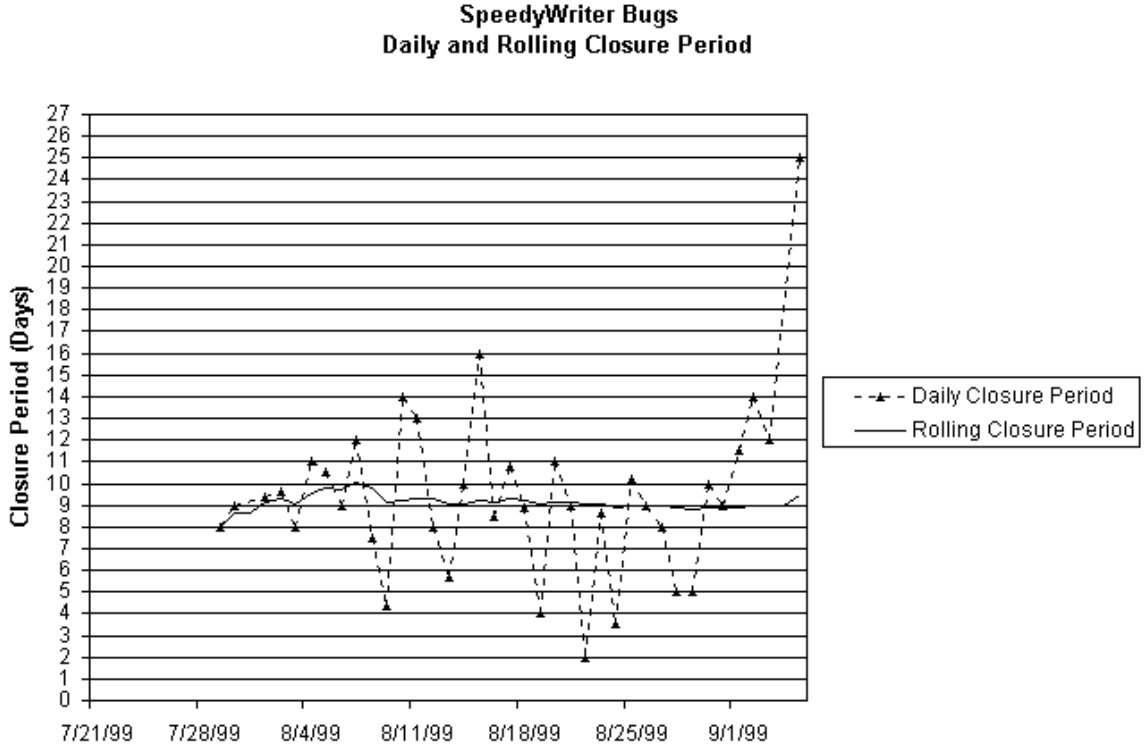



*Figure 5: SpeedyWriter closure period*

Calculating the closure periods, as I mentioned above, is a little tricky. The process involves two separate worksheets in Excel. In the first worksheet, calculate the closure period for each bug report. The first step is to put all the bug report IDs in the left-most column on the worksheet. The closure period for a given bug is `N/A` if the report is not closed. Figure 6 shows the upper third of the Excel worksheet that calculates this number for each bug, including the formula (in the formula bar, preceded with an "=" sign, directly above the worksheet).

Microsoft Excel - Good SW OC RC SS CG Analysis.xls

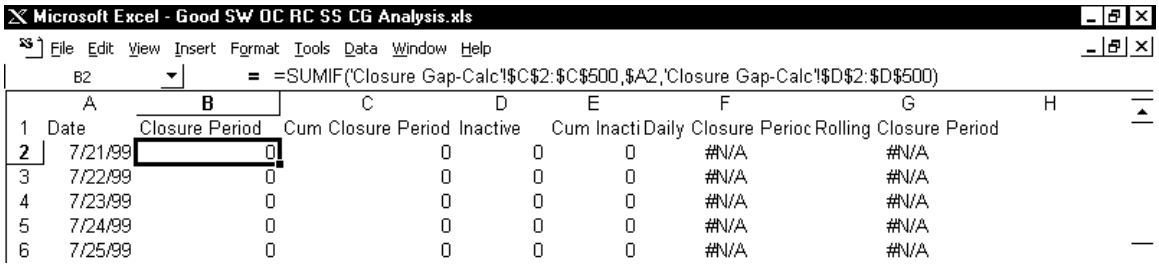
File Edit View Insert Format Tools Data Window Help

D2 = =IF(B2>0,IF(C2>0,IF(ISNUMBER(C2),C2-B2,NA()),NA()),NA())

| | A | B | C | D | E | F | G | H | I | J |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Bug ID | Date Opened | Close Date | Closure Gap | | | | | | |
| 2 | 1 | 7/21/99 | 8/3/99 | 13 | | | | | | |
| 3 | 2 | 7/21/99 | 7/30/99 | 9 | | | | | | |
| 4 | 3 | 7/24/99 | 8/1/99 | 8 | | | | | | |
| 5 | 4 | 7/23/99 | 8/2/99 | 10 | | | | | | |
| 6 | 5 | 7/24/99 | 8/1/99 | 8 | | | | | | |

*Figure 6: Calculating closure period for each bug report.*

On the second worksheet, calculate the daily and rolling closure periods. First, set up the left-most column with all the dates from the start of test execution. Then, using the `SUMIF()` formula, add the closure periods (from the previous worksheet) that correspond to a particular closure date. (This number is zero if no bug report closed on that day.) Then, using the `COUNTIF()` formula, find out how many bug reports closed on that day. The remaining calculations entail totaling and dividing the numbers calculated by these two formulas. (See Figures 7 and 8 for the Excel worksheet, with the two formulas appearing in the respective formula fields.) The closure period chart is obtained by plotting the daily closure period and rolling closure period columns against the dates in the first column.

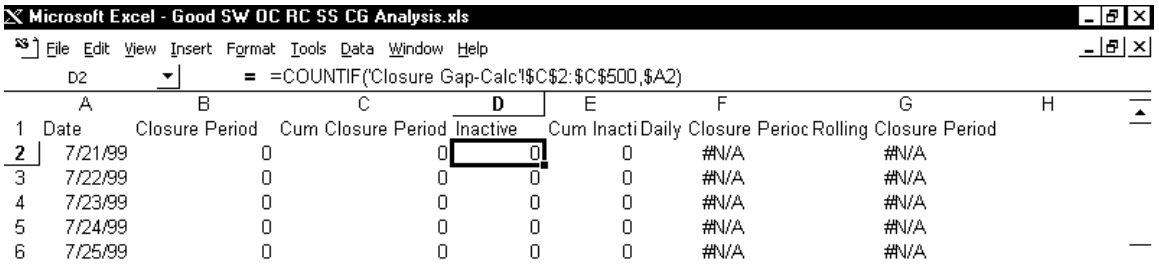

Microsoft Excel - Good SW OC RC SS CG Analysis.xls

File Edit View Insert Format Tools Data Window Help

B2 = =SUMIF('Closure Gap-Calc'!$C$2:$C$500,$A2,'Closure Gap-Calc'!$D$2:$D$500)

| | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| 1 | Date | Closure Period | Cum Closure Period | Inactive | Cum Inacti | Daily Closure Perioc | Rolling Closure Period | |
| 2 | 7/21/99 | 0 | 0 | 0 | 0 | #N/A | #N/A | |
| 3 | 7/22/99 | 0 | 0 | 0 | 0 | #N/A | #N/A | |
| 4 | 7/23/99 | 0 | 0 | 0 | 0 | #N/A | #N/A | |
| 5 | 7/24/99 | 0 | 0 | 0 | 0 | #N/A | #N/A | |
| 6 | 7/25/99 | 0 | 0 | 0 | 0 | #N/A | #N/A | |

*Figure 7: Determining total closure periods by date*


Microsoft Excel - Good SW OC RC SS CG Analysis.xls

File Edit View Insert Format Tools Data Window Help

D2 = =COUNTIF('Closure Gap-Calc'!$C$2:$C$500,$A2)

| | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| 1 | Date | Closure Period | Cum Closure Period | Inactive | Cum Inacti | Daily Closure Perioc | Rolling Closure Period | |
| 2 | 7/21/99 | 0 | 0 | 0 | 0 | #N/A | #N/A | |
| 3 | 7/22/99 | 0 | 0 | 0 | 0 | #N/A | #N/A | |
| 4 | 7/23/99 | 0 | 0 | 0 | 0 | #N/A | #N/A | |
| 5 | 7/24/99 | 0 | 0 | 0 | 0 | #N/A | #N/A | |
| 6 | 7/25/99 | 0 | 0 | 0 | 0 | #N/A | #N/A | |

*Figure 8: Determining bug reports closed by date*

# Root Cause

The third chart breaks down the root causes for all closed bugs. Root cause data is most interesting in the aggregate. Listing the closure of one bug with a specific root cause may not mean much, but seeing the breakdown for a hundred—or a thousand—bugs can tell an engaging story. Figure 9 presents a root cause breakdown for SpeedyWriter, showing the contribution of each type of error to the total number of bugs found and fixed so far. As you might imagine, a chart such as the one in Figure 9 grabs management's attention more effectively than a table.

Capturing root cause data has both short-term and long-term benefits. In the short-term, developers and project management can use the root cause data to course-correct. If many bugs arise from the lack of intelligible specifications, clarifying what the system should do may prevent some of these problems as the project moves forward. In the longer term, for each subsequent project, the project team can use the root cause information from previous projects to improve their process.

The chart shown uses industry-standard categories to group the data. You can choose different categories, you can use these, or you may have to accept the classifications offered by your bug tracking system. The advantage of using industry-standard classes is

that you can compare your project and company results with those obtained by others in the software industry.

You can easily prepare the chart shown in figure 9. The `COUNTIF()` formula allows you to tally the bug reports whose root causes fit into each category. The Excel chart wizard converts this into a pie chart automatically.

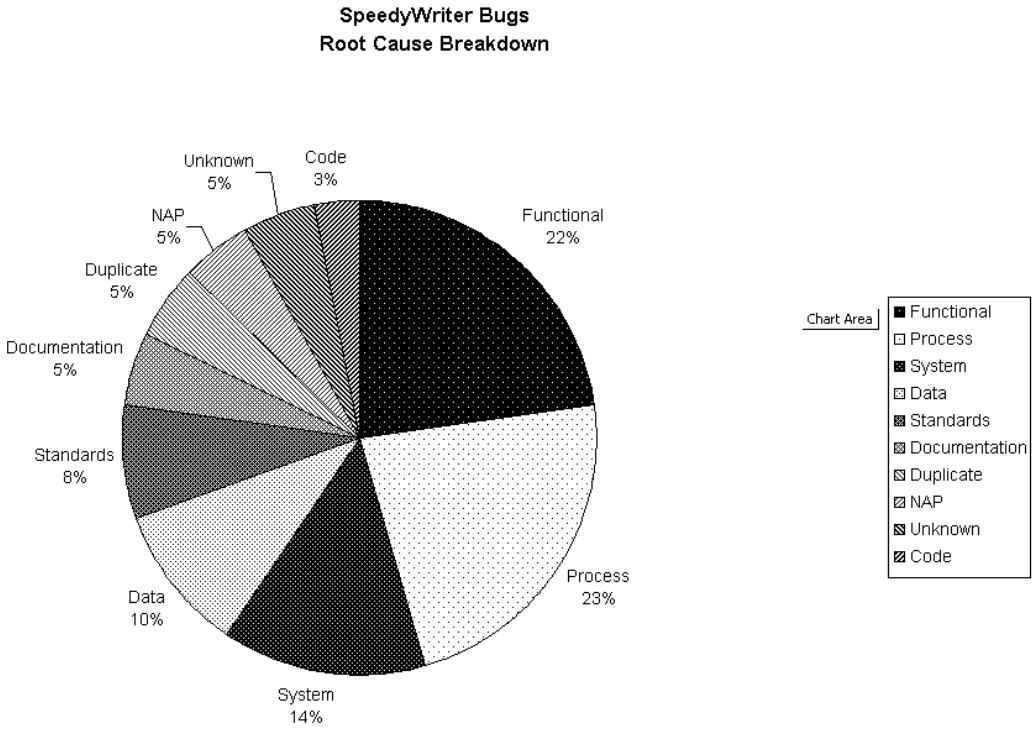



*Figure 9: SpeedyWriter root cause breakdown*

# Subsystem

Like the root cause breakdown, the subsystem breakdown is a simple chart with an important message: it tells you which subsystems experience the most bugs. Since this is usually closely correlated with the subsystem in which the bug exists, you can draw conclusions about which subsystems have the most bugs. This allows the project manager to focus process and product improvement. It's useful to format this as a Pareto chart, as shown in Figure 10, because generally two or three subsystems suffer the most problems. You can use a subsystem chart in the same way you use a root cause chart, to focus process and product improvement efforts. The fact that the user interface and the edit engine account for two out of every three bugs found in SpeedyWriter, for instance, indicates that an effort to make fewer mistakes in these areas would pay off handsomely. In addition, if you are dividing your test effort evenly among the six subsystems, you should consider spending most of your testers' time on these two problem areas. This conclusion might seem counterintuitive—after all, if you didn't find many bugs in the other four subsystems, maybe you should have spent more time looking in those four categories. And you certainly should do this if field-reporting problems indicate a disproportionate number of test escapes in these four areas. However, it is usually the case that where you find many bugs, you will find more bugs. On the five projects

mentioned, the top two subsystems accounted for more than half of the bugs every time, in two cases almost two-thirds, and once almost three-quarters.

Like the root cause chart, the subsystem breakdown is easy to create. On a separate worksheet, use the `COUNTIF()` formula to compute the number of bugs in each subsystem. Sort the list by count so that the most frequently reported subsystem comes first. Then, use another column in the same worksheet to accumulate an overall count for the upper curve on the chart. The Excel chart wizard will do the rest.

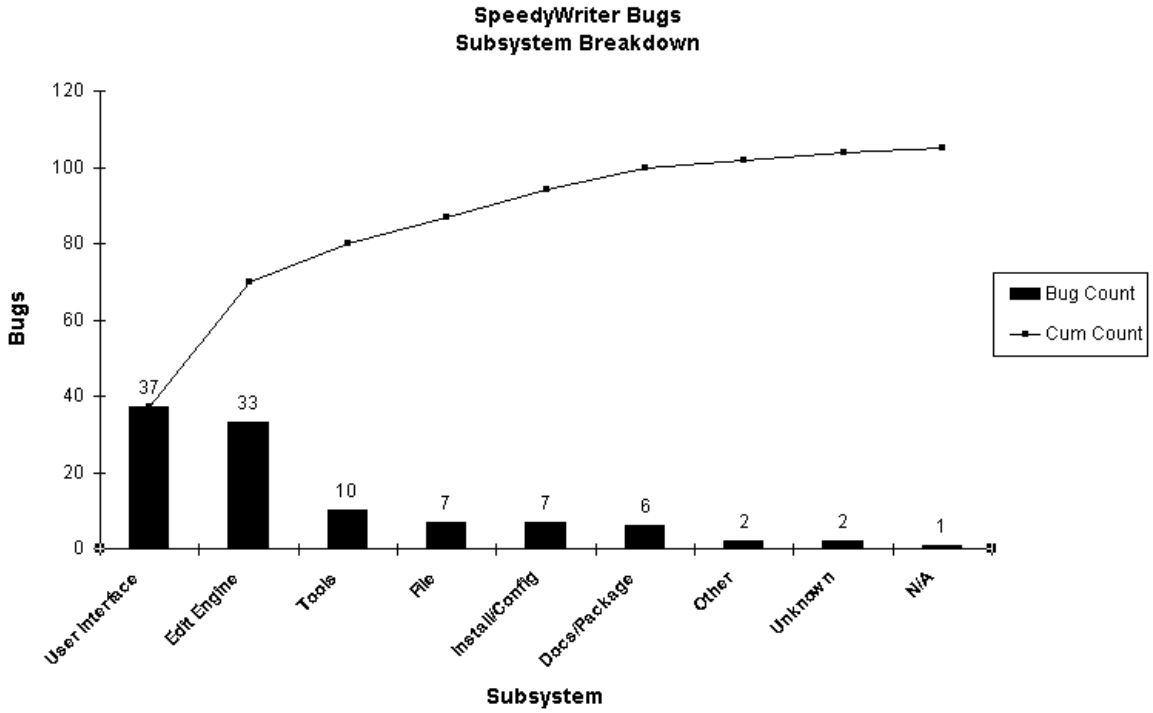



*Figure 10: SpeedyWriter subsystem breakdown*

# Points and Pitfalls

As you accumulate historical bug report data—on both good and bad projects—you can compare charts from your current projects with charts from previous projects. Such comparisons can be enlightening. Even within a single class of projects such as laptop computer development, I have seen variances of 600 percent in the total number of bugs found. Beyond the simple totals, the shapes of the curves can also differ. If you use these four charts consistently across a few projects, you will soon recognize "virtuous" and "evil" curves.

Avoid blind faith in your charts, though. One key assumption of the opened/closed chart is *ceteris paribus* (all things held equal). You can arbitrarily flatten any cumulative opened curve by stopping the test effort. You can cause the opened and closed curves to converge by deferring bugs rather than fixing them. You can spoof opened and closed dates in the database to make the opened/closed chart fit any profile you choose. You can easily rig the closure period chart, too. Mass deferral of stale bug reports, recording phony opened and closed dates, opening new bug reports rather than reopening existing

ones when fixes fail in confirmation testing, and other manipulation of the opened and closed dates will defeat your purpose. Finally, carelessness when assigning subsystems and root causes to bugs renders these charts worse than pointless. Nothing good can come of making decisions based on phony data.

Similar cautions apply to any other analysis of defect or test data. For your analyses to have meaning, the underlying data must be accurate, complete, and free from gerrymandering. Only honest data yields worthwhile information.

Finally, for the closure gap and root cause charts, keep in mind that the charts represent a subset of the overall bug data available. Only closed bug reports count in these charts. In the case of the closure period, that means that the chart probably understates the true daily and rolling turnaround metrics. The extent of the likely error in this chart depends on the proportion of the bug reports that remain open at the moment of concern. The same argument applies to the root cause chart. If fifty or more percent of the bug reports remain open, the root causes reported may not represent accurately the root causes of all the bugs in the product. The opened/closed and subsystem charts do not suffer from this problem to the same extent, being based on all the bugs reported to date, but you are still managing from an incomplete data set until the project is over. Be careful when using incomplete measurements to fine-tune processes in the midst of development or test execution.

# Conclusion

In this paper, I have introduced (or re-introduced) you to four charts I find valuable for managing test projects. The perspectives offered by each chart are impossible to obtain by looking through a stack of bug reports. The higher level of abstraction is enabled by using analysis and charting tools to view certain key variables graphically. This abstraction along carefully chosen data dimensions is what makes these charts good "dashboard" indicators of test and development project performance.

The reports, charts, tables, and forms presented in this chapter are just starters. With a little imagination, you can extract all sorts of useful data and reports from a bug tracking database. Start with simple tasks, learn the tool, and then expand it to meet your needs. For day-to-day management, however, these four charts—or your own enhancements of them—may well suffice. A cluttered dashboard makes it hard to focus on any one key indicator.

# Recommended Readings

Rex Black: *Managing the Testing Process* (1999), Microsoft Press, Seattle, WA.

Boris Beizer: *Software System Testing and Quality Assurance* (1996), International Thomson Computer Press, Boston, MA.

Kaoru Ishikawa: *Guide to Quality Control* (1986), Asian Productivity Organization, Tokyo.

Stephen Kan: *Metrics and Models in Software Quality Engineering* (1995), Addison-Wesley, Reading, MA.

# Biography

Rex Black has spent almost seventeen years in the computer industry, with fourteen years in testing and quality assurance. He is the President and Principal Consultant of Rex

Black Consulting Services, Inc., an international software and hardware testing and quality assurance consultancy (www.rexblackconsulting.com). His clients include Dell, SunSoft, Hitachi, Motorola, Pacific Bell, GE Capital, Tatung, IMG, Renaissance Worldwide, DataRace, Omegabyte, Omnipoint, TeleSource, Strategic Forecasting, and Clarion. He recently completed *Managing the Testing Process*, published by Microsoft Press in its Best Practices series. Mr. Black holds a BS Degree in Computer Science and Engineering from UCLA, and belongs to the Association for Computer Machinery and the American Society for Quality.