# 16 *Software Release*

Driving the software to a releasable condition at the end of each stage is essential to managing the risks of unsuccessful integration and poor quality. Determining whether the software is good enough to release is difficult to do intuitively. Fortunately, several simple statistical techniques can help with that determination. The release stage can be a hectic time, and the use of a Release Checklist helps avoid problems.

As each stage winds to a close, the project team will either symbolically or literally release the software. Whether symbolic or literal, driving the software to a releasable state at the end of each stage is important. The defect count should be brought down to a point where the software could be released to the public, fit-and-finish issues should be addressed, the user documentation should be brought into alignment with the as-built software, and so on.

## Treating Releases Seriously

Overlapping the release phase of one stage with the detailed design phase of the next stage is often a good idea. On a project that doesn't use design and code reviews, developers are held hostage by defect corrections during the release stage. On a project that has done a good job of assuring quality throughout the stage, there won't be enough release-related work to keep developers busy full time, and they will be eager to move on to the next stage.

There are strong temptations to treat part-time release work as a secondary priority. Developers will be more eager to begin work in new areas than to fix problems in old areas. Making progress on new designs and implementations will seem more productive than fixing minor problems with old work.

*The entire project team should treat driving the software to a releasable state at the end of each stage as its top priority.*

The success of the staged delivery approach relies on bringing the software to a releasable quality level and embracing all the extra quality assurance and development work that that entails. Bringing the software to a releasable condition eliminates dark corners in which unforeseen work can accumulate, improving status visibility. If the release phase of a particular stage is allowed to drag on for weeks or months while most of the project team has moved on to the next stage, the ability to determine the project's true status will be lost.

Driving to a releasable state also eliminates the places where insidious quality problems can hide. Without periodically raising the software's quality to a releasable level, the software will begin a slow slide toward low quality, whence it may never return.

I audited a project on which the developers had originally planned to deliver the software in stages. As they approached the end of Stage 1, they decided that they didn't have time to drive the software to a releasable condition, so they moved directly into the development work for Stage 2. By the time my audit team reviewed that project's progress, the project was months behind schedule, mostly because the developers were stuck in an extended test-debug-correct-test cycle. Every defect they fixed seemed to give rise to at least one more defect.

The root of the project's problem was that it had accumulated a large mass of low quality code. When developers added new code, they couldn't tell whether new defects originated from the new code or from the low quality old code. That dramatically increased the time required to debug problems, and made their corrections more error prone. The team finally worked its way out of the situation by calling a complete halt to new code development and focusing the developers solely on fixing defects for more than a month.

The developers' decision at the end of Stage 1 that they "didn't have time" to drive the software to a releasable state was one of the most costly decisions they could have made. They probably were behind schedule when they made that decision. But their decision ultimately put them further behind schedule. If they had stuck to their original plan and had driven their software to a releasable condition at the end of Stage 1, they would have reduced their subsequent test, debug, and correction efforts by a huge factor.

Developers can begin working on the detailed design for the next stage during the release phase of the current stage, but they must be ready to drop their design work at a moment's notice to correct defects detected in the previous stage's work.

## When to Release

The question of whether to release software is a treacherous one. The answer must teeter on the line between releasing poor quality software early and releasing high quality software late. The questions of "Is the software good

enough to release now?" and "When will the software be good enough to release?" can become critical to a company's survival. Several techniques can help you base the answers to these questions on a firmer footing than can the instinctive guesses that are sometimes used.

## Defect Counts

At the most basic level, defect counts give you a quantitative handle on how much work the project team has to do before it can release the software. You can get a summary of the number of remaining defects remaining in order of priority: "2 critical defects, 8 serious defects, 147 cosmetic defects," and so on.

By comparing the number of new defects to the number of defects resolved each week, you can determine how close the project is to completion. If the number of new defects in a particular week exceeds the number of defects resolved that week, the project still has miles to go. Figure 16-1 shows an "open defects" graph, which tracks the status of defects.

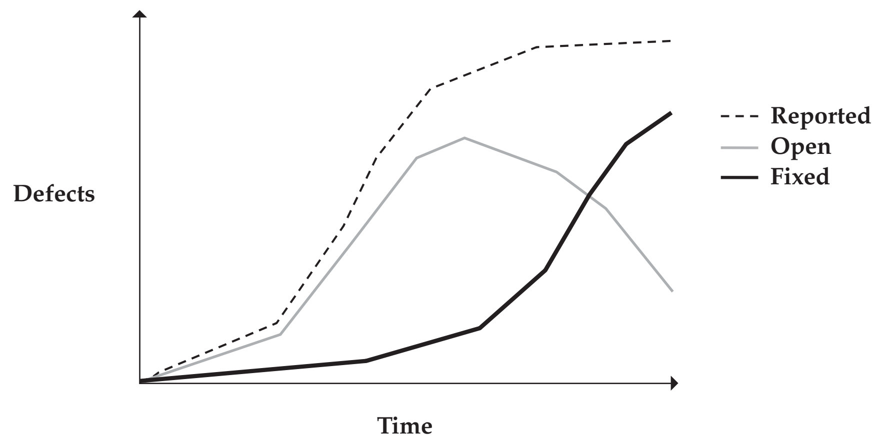


Figure 16-1 *Example of an "open defects" graph. Making this graph public emphasizes that controlling defects is a high priority and helps to keep potential quality problems under control.*

If the project's quality level is under control and the project is making progress toward completion, the number of open defects should generally trend downward after the middle of the project and then remain low. The point at which the "fixed" defects line crosses the "open" defects line is

psychologically significant because it indicates that defects are being corrected faster than they are being found. If the project's quality level is out of control and the project is thrashing (not making any real progress toward completion), you might see a steadily increasing number of open defects. This suggests that steps need to be taken to improve the quality of the existing designs and code before adding more new functionality.

### Statistics on Effort per Defect

The data on time required to fix defects categorized by type of defect will provide a basis for estimating remaining defect-correction work on this and future projects. When you collect this information, by the middle of the project you'll be able to say things like, "The project has 230 open defects, and the developers have been averaging 3 hours per defect correction, so the project has approximately 700 hours of defect correction activity remaining."

The data on phases in which defects are detected and corrected also gives you a measure of the efficiency of the development process. If 95 percent of the defects are detected in the same phase they were created, the project has a very efficient process. If 95 percent of the defects are detected one or more phases after the phase in which they were created, the project has a lot of room for improvement.

### Defect Density Prediction

One of the easiest ways to judge whether a program is ready to release is to measure its defect density—the number of defects per line of code. Suppose that the first version of your software, GigaTron 1.0, consisted of 100,000 lines of code, that the quality assurance group detected 650 defects prior to the software's release, and that another 50 defects were reported after the software was released. The software therefore had a lifetime defect count of 700 defects, and a defect density of 7 defects per thousand lines of code (KLOC).

Suppose that GigaTron 2.0 consisted of 50,000 additional lines of code, that QA detected 400 defects prior to release, and another 75 after release. The total defect density of that release would be 475 total defects divided by 50,000 new lines of code, or 9.5 defects per KLOC.

Now suppose that you're trying to decide whether the GigaTron 3.0 has been tested enough to release. It consists of 100,000 new lines of code, and QA has detected 600 defects so far, or 6 defects per KLOC. Unless you have a good reason to think that the development team's development process has

improved with this project, your experience should lead you to expect between 7 and 10 defects per KLOC. The number of defects the project team should attempt to find will vary depending on the level of quality you're aiming for. If you want to remove 95 percent of all defects before releasing the software, the project team would need to detect somewhere between 650 and 950 prerelease defects. This technique suggests that the software is not quite ready to release.

The more historical project data you have, the more confident you can be in the prerelease defect density targets. If you have data from only two projects and the range is as broad as 7 to 10 defects per KLOC, that leaves a lot of wiggle room for an expert judgment about whether the third project will be more like the first or the second. But if you've tracked defect data for 10 projects and found that their average lifetime defect rate is 7.4 defects per KLOC with a standard deviation of 0.4 defects, you have a great deal of guidance indeed.

## Defect Pooling

Another simple defect prediction technique is to separate defect reports into two pools. Call them Pool A and Pool B. The testing team then tracks the defects in these two pools separately. The distinction between the two pools is essentially arbitrary. You could split the testing team down the middle and put half of its reported defects into one pool, half into the other. It doesn't really matter how you make the division as long as both reporting pools operate independently and both test the full scope of the software.

Once you create a distinction between the two pools, the testing team tracks the number of defects reported in Pool A, the number in Pool B—and here's the important part—the number of defects that are reported in both Pool A and Pool B. The number of unique defects reported at any given time is this:

$$Defects_{Unique} = Defects_A + Defects_B - Defects_{A\&B}$$

The number of total defects can then be estimated by using this simple formula:

$$Defects_{Total} = (Defects_A * Defects_B) / Defects_{A\&B}$$

If the GigaTron 3.0 project has 400 defects in Pool A, 350 defects in Pool B, and 150 of the defects in both pools, as is shown in Figure 16-2, the number of unique defects detected would be 400 + 350 – 150 = 600. The approximate number of total defects would be (400 * 350) / 150 = 933. This technique

suggests that there are approximately 333 defects yet to be detected (about a third of the estimated total defects). Use of this technique reveals that quality assurance on this example project still has a long way to go.

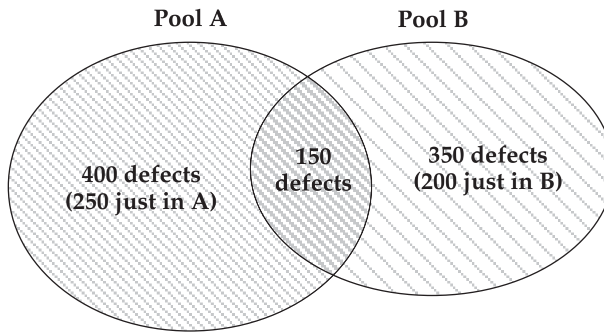


FIGURE 16-2 *Defect pooling. The number of unique defects can be estimated based on the overlap of defects reported in the separate pools.*

The defect pooling technique involves a significant amount of overhead in keeping track of two separate lists of defects and identifying the defects that are common to both lists. It also involves the overhead of covering the entire scope of the software with two independent testing groups. Because of the overhead involved, this technique is best suited to projects that need to be especially accurate in determining their remaining defects prior to release.

## DEFECT SEEDING

As Figure 16-3 on the next page suggests, defect seeding is inspired by a well-developed statistical technique in which a sample from a population is extracted and used to estimate the total population. For example, to estimate the number of fish in a pond, biologists would tag a certain number of fish and release them back into the pond. They would then capture a sample of fish and compare the number of tagged and untagged fish that were captured to estimate the total number of fish in the pond.

Defect seeding is a practice in which defects are intentionally inserted into a program by one group for detection by another group. The ratio of the number of seeded defects detected to the total number of defects seeded provides a rough idea of the total number of program defects that have been detected.

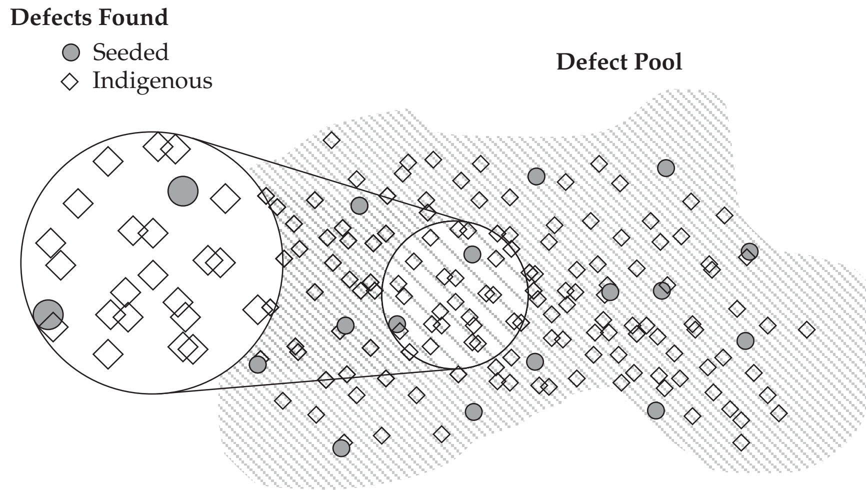


FIGURE 16-3 *Defect seeding. Defects can be estimated based on the ratio of seeded defects found to indigenous defects found.*

Suppose that on GigaTron 3.0, the development team intentionally seeded the program with 50 errors. For best effect, the team's seeded errors should be designed to cover the full breadth of the software's functionality, and they should also cover the full range of severities—from crashing errors to cosmetic errors.

Suppose that at a point in the project when you believe testing to be almost complete, you look at the seeded defect report. You find that 31 seeded defects and 600 indigenous defects have been reported. You can estimate the total number of defects with this formula:

$$Defects_{Total} = (Defects_{SeededDefectsPlanted} \,/\, Defects_{SeededDefectsFound}) * Defects_{FoundSoFar}$$

According to this formula, GigaTron 3.0 has approximately this number of total defects: (50 / 31) * 600 = 968. Almost 400 defects have yet to be detected.

To use defect seeding, the developers must seed the defects prior to the beginning of the tests whose effectiveness you want to ascertain. If the testing team uses manual methods and has no systematic way of covering the

same testing ground twice, the developers should seed defects before that testing begins. If the testing team uses fully automated regression tests, the developers can seed defects virtually any time to ascertain the number of indigenous defects the automated tests have detected.

A common problem with defect seeding programs is forgetting to remove the seeded defects. Another common problem is that removing poorly designed seeded defects can introduce new errors. To prevent these problems, be sure to remove all seeded defects prior to final system testing and software release. Some projects require that seeded defects be kept extremely simple, and they allow introduction of only those seeded defects that can be created by adding exactly one line of code.

## Defect Modeling

With software defects, no news is usually bad news. If the project has reached a late stage with few defects reported, there is a natural tendency to think, "We finally got it right and created a program with almost no defects!" In reality, no news is more often the result of insufficient testing than of superlative development practices.

Some of the more sophisticated software project estimation and control tools[1] contain defect modeling functionality that can predict the number of defects you should expect to find at each stage of a project . By comparing the number of defects actually detected to the number predicted, you can assess whether the project is keeping up with defect detection or lagging behind.

## The Release Decision

If you follow the practices described in this book, you'll have solid information upon which to base the decision about whether software is ready to release. These sources of information include the following:

- Code growth statistics and graph (refer to Figure 5-6 on page 62)
- Detailed list of binary miniature milestones completed
- List of raw defects from the defect tracking system
- Cumulative defect statistics and graph (refer to Figure 16-1 on page 224)

1. For a current list of these tools, see the *Survival Guide* Web site.

- Effort-per-defect statistics
- Defect density prediction
- Defect pooling
- Defect seeding
- Defect modeling

Evaluating combinations of these readiness indicators will give you more confidence than you could have from evaluating any of the techniques individually. Examining defect density alone on GigaTron 3.0 suggested that you should expect 700 to 1000 total lifetime defects, and the project team should remove 650 to 950 defects before software release to achieve 95 percent prerelease defect removal. If the project team had detected 600 defects, the defect density information alone might have led you to declare the software "almost ready to release." But defect pooling analysis estimated that GigaTron 3.0 will produce approximately 933 total defects. Comparing the results of those two techniques suggests that you should expect a total defect count toward the high end of the defect density range instead of the low end. Because the defect seeding technique also estimated a total number of defects in the 900s, it seems evident that GigaTron 3.0 will exhibit a relatively large total number of defects and that the project should continue testing.

### Defect Tracking and Communication

Publicizing the kind of status and quality information discussed in this section helps to keep the project on track. The project team should post defect summary information in a public place, such as in the project break room, on the project manager's office door, or on a project intranet Web page.

## Release Checklist

At the end of a stage, even on the best projects, the most serious errors committed are often simple oversights. People want their work to be done, feel that it is done, and have a tendency to skip seemingly obvious details.

Very early in my software career I was the project manager for an insurance consulting company that provided insurance rate quotation programs for its clients. These were very simple programs by today's standards,

each involving no more than about 3 staff-months of effort to create, and most requiring considerably less development time. Even with these simple programs, we managed to run into the most common problem with software releases—simply forgetting some of the things we knew we needed to do before we released the software to our clients! As a result of our hard-won experience, we created a Release Checklist, which included items like this:

- Make exact duplicates of diskettes before sending them out.
- Make a list of all the people receiving the program and the number of diskettes mailed to them.
- Put postage on packages sent to clients.

If you read between the lines, you might guess that at one time we didn't keep exact copies of the programs we mailed out and couldn't reproduce problems clients reported. We sometimes didn't know which clients had received which programs, and one time we even forgot to include postage on a program we sent to a client.

More sophisticated applications require more sophisticated release procedures, but the basis of any release procedure will still be a checklist of activities that need to be done before the software can be released, and that checklist will inevitably consist of many activities that were forgotten at one time or another. A Release Checklist for a simple program might contain only a handful of items. The Release Checklist for an extremely complicated program such as Microsoft Windows 95 might contain 200 items or more.

Table 16-1 (on pages 232–233) shows some of the items that should be included on the Release Checklist for a medium-sized software product that will be released to the general public. The focus of the list is not on testing—by this time it is too late in the project to start worrying about that. The focus is on items that are easily overlooked in the haste to push the software out the door.

If the project team is releasing the software to in-house users, the checklist will look different, but the same idea applies: the checklist should capture the critical release activities the project team doesn't want to forget in the rush to release the new system. The project team should put together Release Checklists for interim releases and for the final release. The different lists will not be identical but will have many points in common.