Sifting through the Software Sandbox

SCM meets QA

Source control—it's not just for tracking changes anymore.

WILLIAM W. WHITE, PERFORCE SOFTWARE

Thanks to modern SCM (software configuration management) systems, when developers work on a codeline they leave behind a trail of clues that can reveal what parts of the code have been modified, when, how, and by whom. From the perspective of QA (quality assurance) and test engineers, is this all just “data,” or is there useful information that can improve the test coverage and overall quality of a product?

TESTING AS USUAL

Test and QA engineers have a variety of tools at their disposal when they set out to assess a software product. They can use functional specifications and user manuals to map out a test plan; design specifications and interviews with developers can provide the information needed to develop functional tests; scripting tools can be used to automate the execution of some of those tests; and code coverage tools can be deployed to identify gaps in the testing procedures. After the test plan has been written and the basic test suite developed and executed, however, much work remains to be done.

Test plans and automated test suites provide some assurance that the features that a software product is designed to provide are present and operate as intended.

As the product grows and evolves, these tools can evolve accordingly, to include additional cases for new or modified features. Throughout the life of the product, they are also invaluable in guarding against regression, the introduction of defects into features that worked correctly in previous releases.

Regression bugs are relatively rare, however, and once a product has been shipped, test plans and automated test suites only occasionally find bugs. What's more, these tools do nothing to help a tester know where bugs are likely to occur, or to help a QA engineer make a realistic assessment of the quality of a product that's about to go out the door. Fortunately, there are other tools that can help.

DEFECT TRACKERS

For decades software developers have used defect-tracking tools to manage information about known bugs and requests for product improvements. Almost all such tools include fields that identify the nature and severity of each problem, as well as such information as when it was recorded and when it was fixed. This minimal information is enough to give testers and QA engineers powerful leverage in assessing the quality of a product.

Sifting through the Software Sandbox

SCM meets QA

The most basic use of this information with respect to testing is in verifying that bugs that were reported to have been fixed really were fixed. This is usually accomplished by generating a list of items whose “fixed” or “closed” date falls within a specified range—for example, after the previous release was shipped, but before the code freeze for the pending release. A tester can then go through that list, following the reproduction steps in each problem description. For each item, if the problem no longer manifests, the tester checks it off and goes on to the next one; if the problem is still present, the item is reopened and corrective steps are taken.

In addition to bug fixes, product updates usually introduce new features—which can sometimes introduce new bugs. To guard against that possibility, the product undergoes a period of intensive testing prior to release, to “shake out” any problems introduced during the development phase. It’s not unusual for a significant number of bugs to surface during this period.

Although QA staff take note of the total number of bugs found, an observation that is perhaps more mean-

ingful is the rate at which those bugs are reported. In a typical case, shown in figure 1, a relatively large number of bugs are “popped” during the first few days of the pre-release testing period, and that number gradually tapers off as the period draws near its scheduled end. A trend that showed a different pattern—for example, a bug rate that remained high, or one that began low—would suggest that there may be problems with either the product or the testing process, and might warrant the delay of a release to allow for more testing and fixing of bugs.

After a product is released, the information from a defect-tracking system can be used in several ways to assess the quality of the ongoing maintenance effort. For example, it would be reasonable to expect that serious software problems encountered by customers would be corrected in a timely manner. One simple way to find out if this is happening is to plot the number of new bugs reported per week against the number of bugs fixed during the same period, as shown in figure 2. Ideally, the trend would show the fix rate following the rate of newly reported bugs very closely, especially for high-priority issues. Note, too, that the overall trend shown in this figure should mirror that drawn during the pre-release testing: as the product matures, you would expect fewer new bugs to be reported from the field.

Any number of additional metrics can be obtained from an analysis of the information in a defect-tracking system. (See Stephen Kan’s *Metrics and Models in Software Quality Engineering*, Addison-Wesley Professional, 1995,

for some other examples.)

In general, however, these tools are limited by their “black-box” view of the software against which they are applied. While it may be easy to count the number of problems reported, and the tool may allow you to include in the report information about the specific module where the error occurred, most defect-tracking tools can’t identify the exact location within the source code where the fault occurred. This means that a tester needs to rely on intuition and knowledge of the internals of the product

Defect Reporting Rates During Pre-release Testing

FIG 1

when trying to determine whether a bug has affected other features. Since the effects of a bug can sometimes be widespread, and may surface far from the place where they were initially observed, the information available from typical defect-tracking applications may not be entirely adequate for use as a guide in testing.

FIRST-GENERATION SCM TOOLS: DISCRETE PER-FILE CHANGE TRACKING

As is the case with defect-tracking systems, developers have, since at least the 1970s, used tools to manage changes made to source code. The focus of these tools has been on their usefulness to programmers who develop and maintain code, but testing and QA professionals have benefited, too.

Mining QA-related information from simple per-file tracking systems such as RCS (revision control system) or CVS (concurrent versions system) is possible if testers have access to the complete archive files for a product, and if they are willing and able to dig in and develop the scripts needed to extract and summarize the relevant data. This work also requires a measure of intimacy with the source code that may be beyond the comfort zone of many managers or testing staff. More to the point, the degree of access to the source archives that is required might strain the comfort level of the executive staff at many software companies.

The kind of information that a QA engineer could glean from RCS, for example, would be a very rough

measure of the volatility of a codeline. This could be obtained by counting the number of file changes that have occurred between the last customer ship and the most recent build. If the last customer ship were delivered on August 1, 2004, and the most recent release candidate build were done on September 22, a shell command such as the following would tell you how many changes had affected files in the current directory:

```
% rlog -d"2004-08-01<2004-09-22" *,v | grep ^revision | wc
405 810 5556
```

In this example, you can see that 405 change submissions were made in the current directory between August 1 and September 22. This doesn't tell you anything about how extensive the changes were, but it does, at least, give you one hard fact from which to work.

Aside from the complexity of the changes, is 405 a large number or a small one? Running the same command in each directory of your source tree will give you part of the answer—any directory with a high number of changes will be of interest. The rest of the answer lies in how many files in each directory are under source control and the relationship between the number of files in each directory and the number of change submissions. Files that have been modified many times can be expected to be less reliable than files with few modifications.

If, for example, the average ratio of change submissions to files throughout your source tree turned out to be 3:1, but one of your directories had a ratio of 20:1, then the product modules associated with that directory would be a good place to look for newly introduced bugs. Likewise, if one directory had 200 recorded changes, compared with an average of 10 for other directories, then that directory would warrant scrutiny even if it contained a large number of files, and the ratio of changes to files was relatively low.

Stepping through RCS commands by hand and doing the math isn't difficult, but it would be tedious and time consuming for a typical software product with a large and complex source tree. Fortunately, it's simple to

Sifting through the Software Sandbox

SCM meets QA

write scripts to handle much of the work, and more complex scripts could be crafted to process all of the information and summarize the results in a table similar to that shown in figure 3. In testing, emphasis should be given to product modules associated with directory sub1 in figure 3 because of the high number of change submissions there. Emphasis should also be given to modules associated with sub3 because of the high number of changes made per file.

NEXT-GENERATION SCM TOOLS: CHANGESETS AND CHANGE NOTIFICATION

Modern SCM systems include other features that can be of use to people involved in testing and QA. One of these is the ability to group a set of file changes and submit them as a single entity. These groups are usually referred to as *changesets* or *changelists*. Another useful feature is the ability to request notification whenever changes occur in specified parts of the source tree.

Changelists are of obvious value to developers and porting engineers who might, for example, want to implement a bug fix on a new platform or back out a change that turned out to have undesirable side effects, since it allows them to immediately see the complete

Changes per directory and ratio of change submissions to file count

Directory	Files	Changes	Ratio
/depot/project/sub1	100	1000	10.0
/depot/project/sub2	20	40	2.0
/depot/project/sub3	10	200	20.0
/depot/project/sub4	30	40	1.3
/depot/project/sub5	20	10	0.5

FIG 3

set of files affected by the change. They also have the significant advantage of providing a single, obvious place where a complete description of the change can be recorded—there's no need to duplicate this information for every file involved, nor is there a need to establish a formal procedure that every developer must follow to assure that the people downstream are able to find all of the pertinent information.

The descriptions entered into changelists turn out to be useful in focusing the testing effort, especially in the weeks leading up to a product release—provided, of course, that the developers who submit the changes are reasonably thorough in describing them.

A simple list of the descriptions for all of the changes that were made since the previous release of a product,

Descriptions of the 100 Most Recent Changes Made Prior to September 1, 2004

% p4 changes -l -m 100 ...@2004/09/01

Change 12345 on 2004/08/31 by www@www_client

Remove 2 obsolete files and change the way the object list is written. Instead of leaving it commented out, enclose it in "#ifdef DEBUG" tags.

Change 12034 on 2004/08/30 by joe@joes_client

Add a check in foo.cpp for the case where i is null. Fix for bug #54321.

Change 11043 on 2004/08/29 by alice@some_client

Add a new class to Foo.h to encapsulate the display presentation that we use in dialogs. This will help reduce duplicate code.

[...]

These changes affect files in the current directory or any of its subdirectories (Perforce syntax).

FIG 4

for example, can amount to a system-generated test plan for the upcoming release, as shown in figure 4. It will be unpolished and its organization will be purely chronological, and it will probably contain a significant amount of information that isn't particularly helpful in the context of testing. Nonetheless, a patient tester can sift through this data and emerge with a pretty clear picture of what has changed and what needs special attention. Combined with a set of regression tests to catch any adverse side effects of recent changes, this is a powerful tool.

Likewise, automatic change notification can instantly alert interested parties to code changes that might affect their own work. It can allow a supervisor to monitor, in realtime, updates made to the code, and it can tell testers and QA engineers the moment a new feature or bug fix appears in the product and is ready to be tested.

Some modern SCM systems either include defect trackers or allow you to integrate third-party products such as Bugzilla, or both. An integrated defect-tracking system can automatically correlate bug fixes with change descriptions. This allows you to instantly compare a bug report, typically written from the customer's point of view, against the developer's description of the underlying problem and the steps taken to correct it. A system that includes change notification and that ties bug reports to changelists lets you know instantly when the status of a problem of interest to you is updated. It also lets you zoom in on the details of the change, even to the point of examining the exact coding modifications that were made to each file that was affected by it. This combination of features can allow testers to keep abreast of bug fixes, new feature implementation, and any other changes that occur in a product.

WHAT'S STILL AHEAD?

SCM systems have been evolving for decades, and their ability to gather and store data pertaining to changes is impressive. Until fairly recently, however, most systems have relied on command-line interfaces to communicate that information to their users. This has, for the most part, been adequate

for developers, who tend to be interested in very detailed information about a specific file or change, but it has meant that organizations such as QA, which need to highlight trends by analyzing and summarizing information, are forced to develop scripts that call the command-line interface, gather the raw data that it emits, and iterate through it. Once the data has been digested in this way, the final results can be formatted in tables or charts that convey the meaning behind the data. Third-party tools such as spreadsheets or Crystal Reports are often used for this purpose.

The good news is that providers of SCM solutions are increasingly offering GUI interfaces for their products. Some are Web-based, some use standard Windows protocols, and others use third-party libraries such as Trolltech's Qt or cross-platform programming tools such as the Java language and its graphical extensions to produce applications that run on a wide range of platforms while retaining a look and feel appropriate to each of their supported operating systems. Regardless of their implementation, these interfaces improve the productivity of software developers by removing the requirement that they learn an arcane new set of commands. Instead, developers can simply point and click to navigate through their source code, checking out and editing files, submitting changes, comparing file and folder revisions, and performing all of the standard operations pursuant to getting their work done. Useful as these interfaces are to developers, they hold much more promise for QA and testing staff.

Sifting through the Software Sandbox

SCM meets QA

SCM and defect-tracking systems store vast amounts of data concerning the status and the change history of a software product. For example, all of the information needed to calculate defect arrival and fix patterns is available in many SCM data stores, and this information, once extracted and summarized, represents a useful measure of the effectiveness of a software maintenance team.

Extracting and summarizing this data now involves running a set of scripts that call out to the SCM system's command-line interface to count the number of bugs reported during a specified period of time, then to count the number of bug fixes submitted during the same period. When these totals are gathered for a range of intervals and summarized in tables or charts, a clear picture of a software team's responsiveness to reported problems emerges. Note that the information can be plotted in discrete time intervals, as shown in figure 2, or, more commonly, as the cumulative number of bugs reported and fixed over an extended period of time, as shown in figure 5. You can further refine the information by separating the counts based on bug severity, since you would expect to see an immediate response to more serious bugs, whereas less serious ones might be deferred until a subsequent product release.

Is this good enough? SCM and defect-tracking systems provide the means to gather, store, and extract the information necessary to make measurements like these, and once the scripts have been written, it's a simple enough matter to run them. They can even be scheduled to run automatically on a nightly or a weekly basis, perhaps with the results posted on an internal Web page.

The emergence of GUIs in SCM systems, however, raises the possibility that scripts, schedules, and manual processes could be set aside in favor of realtime, configurable, on-demand reports. Upper management might still need periodic, regularly scheduled summaries, but QA engineers in the trenches and front-line managers often want to know, "How are we doing right now?" Since all of the data needed to produce these reports is directly available to the SCM system, it would be fairly easy for a graphical interface to provide this information whenever it is needed, and in whatever form is desired.

POWERFUL WEAPONS—BECOMING MORE POWERFUL

With modern SCM and defect-tracking systems, QA and testing engineers have powerful weapons to augment the test plans, scripts, and code coverage tools on which they have traditionally relied to assure the quality of software products. They can now be automatically notified the moment an update is made to the status of a bug that they are monitoring, or when a change is submitted to a part of the source tree that concerns them.

Once extracted and summarized, this information can tell a QA engineer or a product manager a great deal about the status, stability, and quality of a software product and can be invaluable in answering the ultimate question: Is it ready to ship? Extracting and summarizing the information from an SCM system, however, is still labor-intensive. Consider, for example, trying to determine defect arrival and fix patterns, or to measure the defect density or the relative volatility in a codeline or a product or a particular subset of the source tree. This process would require QA engineers to develop, deploy, and maintain customized scripts that work in conjunction with the SCM system and/or the defect-tracking tool.

The addition of GUIs to SCM and defect-tracking systems may change all of that. For the first time, it might be feasible to move from systems that merely track defects to systems that provide realtime defect and quality analysis. Imagine the leverage you could have in testing a pending release if you could click a button on a GUI and see a Pareto diagram highlighting the areas in your source tree where the highest number of defects had been reported, or if you could instantly view a chart showing the defect reporting and fixing trends for a product, a subsystem within a product, or an arbitrary subset of your source tree. Or imagine if you could select an option from a context menu and instantly see a measure of the defect density for a section of your code.

The future of QA-oriented features in SCM systems is still being defined, but much of the data is already there, and the possibilities for making use of it through graphical interfaces are exciting to consider. ☺

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

WILLIAM W. WHITE has more than 20 years of experience as an application designer and developer, software engineer, QA engineer, and writer. He currently manages the QA department at Perforce Software.

© 2005 ACM 1542-7730/05/0200