



**What's Wrong with Your Testi**

**Using Source  
Code Manage**

**TO**

**Reduce**

**Redundancy & Find De**

*by William W. White*



ng Strategy?  
ment  
e  
ffects

## Iterative Development

Software developers have long known that it is unrealistic to expect to get a complex, new product exactly right on the first try. Even a relatively simple application incorporates so many details of logic flow and user interaction—and rests on so many assumptions about the underlying problem—that it's all but impossible to avoid tripping on some rare scenario or unexpected condition that wasn't obvious at the outset.

Today, most software is developed using a process that includes mock-ups and prototypes to give designers, developers, and end-users a more complete understanding of what the new system will look like and how it will behave. Often, this prototype code evolves to become the final product as the project progresses. This works well, but it can put the testing staff in the uncomfortable position of holding up the release. When the design and the coding are complete, and the product seems ready to ship, it is hard to understand why it should take so long just to test it.

## Iterative Testing

Testing doesn't have to be the bottleneck that holds up a release, even when the design and development are iterative. The information in your source code management (SCM) system can help your testing evolve along with the design and the code.

It seems simple enough. The developers check their work into the SCM system at the end of each day, and every night an automated script builds a new set of executables for testing. Every morning, the testers arrive at work and pick up the latest executables. No more waiting for a formal hand-off at the end of the development cycle before the testing can begin.

Now you have iterative testing, but do you see the problem with this scenario? The problem is that, early in the project, the software under development is evolving very rapidly. If the testers are going to pick up a new version of this product-in-progress every morning, they're apt to spend a lot of time retracing their steps and retesting the same things they tested a day or a week before. They may even think it would be better to wait until the system stabilizes before starting to test.

## Using SCM to Focus Testing

The challenge for testing, especially in a dynamic environment like this one, is to know what to test. In the early stages of a project, regular meetings of the designers, the developers, and the testing staff will make it clear where the focus should be. But as development proceeds, and as the project gets closer to being functionally complete, it becomes more difficult to know exactly what the developers have been working on. And if you're facing a big, collaborative project, with hundreds of thousands of lines of code and dozens of programmers all working on it at once, it will become increasingly difficult to know what needs to be tested. SCM can help.

## Tracking Change in Real Time

The first and most immediate way that SCM can help is by showing you exactly which files are being changed. As testing professionals, we know that a high percentage of the defects in a system are going to be in the parts of the code that have changed the most.

In the early stages of a project, the programmers will be fo-

cused on implementing substantial new blocks of code, but later on, after the major functionality is complete, the coding effort will begin to diffuse. Instead of working in a concentrated way on a specific feature or set of features, the developers will be fixing defects and tweaking features throughout the product and making changes all over the code base. When this happens, it can be very difficult to know what to test—unless you have a way to monitor what’s changing in the code. This is exactly what your SCM system can do for you.

Most SCM systems can provide this support for testers. Since I work for the company, I will use the Perforce SCM system to illustrate the specifics. The simplest way to keep on top of what has changed is to set up your user account to “review” changes made to files in directories that you’re interested in. A user account contains just a few data fields: a user name, an email address, the date when the account was last updated and when it was last accessed, an optional full name for the user and—here’s the useful piece—an optional list of directories that the user wants to “review.” A typical user profile might look like

```
User:      bruno
Email:    bruno@CompanyX.com
Update:   2005/10/11 13:05:15
Access:   2007/02/03 15:13:55
FullName: Bruno Sammartino
Reviews:
//CompanyX/ProjectY/main/...
           //CompanyX/ProjectY/r06.2/..
```

Figure 1

figure 1.

With this entry in his user profile, bruno will receive email whenever a change is made in the main development line or in the 6.2 release line for ProjectY.

So, at the beginning of the project, the tester sets a `Reviews` entry listing the directories in which the coding for the project will be done. This has two immediate benefits: First, it provides assurance that any changes the programmers make won’t slip by unnoticed, and second, it lets the tester begin to get a feel for what features are implemented in what parts of the code. This becomes very important as the project proceeds. If you’re aware of the relationship between features and the source code that implements them, then you’re more likely to know what to do when you see a code change come through later, even if the description that the submitter entered for the change is cryptic or incomplete.

If you’re able to go back to the developer and say, “Hey, it looks like your fix for the NT handle leak is causing a problem

on Windows Server 2003,” then that developer is likely to feel he’s dealing with someone he can trust, someone who knows what he’s doing—and the relationship between coder and tester just improved a little bit.

## Reviewing Change History

A steady, unsolicited stream of email notifications describing individual change submissions is a useful thing; it keeps you on top of what’s happening in the code you’re testing. But there’s a limit to how helpful this can be. A month into the project, you won’t be able to remember every change that went by, and if a milestone in the project is coming due, you might want to take a step back and review what’s been happening. Your SCM system can help with this, too. It’s easy to generate a report that summarizes changes that have been made to the code base; the Perforce system uses the `p4 changes` command for this.

This lists all the changes that have ever been made to any part of your source code repository—which is probably a little more information than you really want. But it’s easy to narrow the scope. For example, you can limit the list to changes affecting a particular subset of the source tree by specifying a partial directory path:

```
$ p4 changes //CompanyX/ProjectY/main/...
```

The ellipsis at the end of this path tells the server that you want to include any subdirectories that occur beneath the last directory you specified in the path.

You can further narrow the scope by specifying a date range. For example, if you want to see all of the changes made in the “main” directory tree of the ProjectY development branch between November 20, 2006 and November 28, 2006, you can use a command like this:

```
$ p4 changes //CompanyX/ProjectY/main/...2006/11/20,2006/11/28

$ volatility.rb //CompanyX/ProjectY/main/...
Change summary: Most recent 100 changes for //CompanyX/ProjectY/main/...

//CompanyX/ProjectY/main/ModuleA      76.2%
//CompanyX/ProjectY/main/ModuleC      16.7%
//CompanyX/ProjectY/main/ModuleX      5.6%
//CompanyX/ProjectY/main/ModuleY      1.5%

Total files affected: 324
```

Figure 2

This command generates an inventory of test objectives. It’s certainly not a complete list of everything that needs to be tested, but it is a complete list of everything that has changed—and that’s where the defects are likely to be.

You can generate a report like this at any time just by typing a simple command. Or you can set up an automated script that will run the report on whatever schedule makes sense to you. It

```

# -----
# Run the "p4 changes" command to get a list of the target
# changelist numbers.

def run_changes_command( depot_path, changelist_count )
  cmd = sprintf("p4 changes -m %s %s", changelist_count,
    depot_path)
  changes_out = `#{cmd}`
  return changes_out
end
# -----

```

Figure 3

can email you the results or post them to an internal Web page, so you'll have a nice activity summary whenever you need it. Either way, now you have a better grip on what's been changing and where you need to be especially thorough in your testing. Since the output from the `p4 changes` command includes the individual change submission numbers, you can drill down on any change of interest and see as much information as you need about it, right down to the individual lines of the individual files that were updated by any given change.

## Using Scripts to Summarize Change

When you're deep into a big project—one that involves a large number of files and a huge staff of developers—or you're a test manager or project leader responsible for a whole range of products, a stream of email reports that detail individual change submissions would probably be too much data to be useful, and even a list of change submissions keyed to a specific directory tree and date range might be overkill. Often what you really need is a summary.

Using simple scripts, you can mine the information from your SCM system and summarize it in ways that you find useful. For example, if you're managing a staff of testers, and each tester is assigned to a specific product or group of products, it might be useful to know which products have been undergoing a lot of change and which have been relatively stable. Armed with that information, you might decide to ask some of your staff members to set their normal work aside for a little while and help out with the testing of a more volatile product.

## A Script to Analyze Code Volatility

Let's take a look at an example of a simple script that can tell you something about the volatility of your source code.

The Ruby script `volatility.rb` runs a series of commands and produces a report that shows which directories have seen a high rate of change and which have not. The script takes a pa-

rameter in the form of a depot directory path, analyzes all of the recent changes that have affected files in that path, counts the number of files that were touched by each of those changes, and prints a list of the directories containing those files, from the most volatile—the ones whose files have been changed the most often—to the least volatile.

Figure 2 shows what the output from a run of `volatility.rb` might look like.

In the `//CompanyX/ProjectY/main` path in our source tree the last one hundred changes made touched 324 individual files, and 76.2 percent of them

were in `ModuleA` of the project. Clearly, the developers assigned to this module have been busy, and we can assume that it needs to be thoroughly tested. Conversely, we know that any projects

```

$ p4 changes -m 100 //CompanyX/ProjectY/main/...
Change 776 on 2005/01/27 by ines@ines-rose 'Add hooks to simplify debuggin'
Change 775 on 2005/01/27 by gale@gale-cedar 'First set of user documents'
Change 768 on 2005/01/21 by quinn@quinn-dev-azalea 'Add jamgraph image'
Change 766 on 2005/01/21 by quinn@quinn-dev-azalea 'Merge changes'
...

```

Figure 4

whose source directories aren't listed here haven't been changed recently.

Figure 3 is a brief look at how the `volatility.rb` script works.

The `depot_path` parameter is the directory path that we specified on the command line, and the `changelist_count` is the number of recent changes to include in the analysis. If no `changelist_count` is specified on the command line, the script will report the most recent one hundred changes that affected files in the specified directory path.

The raw output from this command is shown in figure 4.

As shown in figure 5, the script then looks at each line of that output, extracts the change number, and calls `p4 describe` (see figure 5).

By default, the output from a `p4 describe` command includes the change description, a list of the files affected by the change, and a block of text in Unix diff format showing the individual changes made to each file affected by the change. Since our script isn't interested in the individual file differences, we exclude that output with the `"-s"` (summary) option. The output for the first change is shown in figure 6.

The `volatility.rb` script captures the file list for each change submission and appends it to an internal array of file paths. That array is then sorted, and the number of occurrences of each file is counted and added to a running total for each directory. Finally, the totals are sorted from highest to lowest, percentages are calculated, and the summary is printed.

```

# -----
# Take one line of output from "p4 changes", run "p4 describe -s"
# on the change number, and peel out the list of filenames
# affected by it.
def get_file_lines( change_line )

  split_line = change_line.split
  p4cmd = "p4 describe -s "
  p4cmd << split_line[1]
  describe_output = `{p4cmd}`
  file_lines = ""
  describe_output.each_line { |theline|
    if theline.index("... //") == 0
      then
        file_lines << theline
      end
    }
  return file_lines
  end
# -----

```

Figure 5

## Other Available Information

This is just one example of how your SCM system can help guide your testing efforts. But if you think about the kind of information that an SCM system manages—all of the details about which files were changed, when, how, and by whom—you see that much of this information is of interest to testers.

If your SCM system is integrated with your defect-tracking system, you have a genuine mother lode of testing and quality data. In addition to identifying which areas in the source code have been changing a lot, which individual files have been modified frequently, and how extensive those modifications have been, you can track defect fixes back to the files where the errors occurred. Then you can pinpoint the parts of your code that have been responsible for higher numbers of defects; those parts are likely to have more defects going forward, too. You could find out how many defects your testing team caught before your last release went out and compare that to the number that your customers subsequently found, and you could compare those numbers from one release to the next and get a sense for whether your processes are improving or going downhill.

Note: One thing to be careful of when mining SCM data is correlating data by developer. Every change record includes the identity of the developer who submitted it, and it can be tempting to wonder which developers were responsible for introducing the most defects—this information isn't obvious, but it can be extracted if you work at it. But, in addition to getting your calculations exactly right, it's important to remember that there can be any number of reasons why one developer might be responsible for more defects than another. Consider that your best developers will probably be assigned to your most

difficult and complex code, and they will probably work faster and produce more output than their more junior colleagues. So, while it might be interesting to know which developers introduced the most defects into your products, and although that information could conceivably be of value, you should be very careful about the conclusions you draw from it.

## Conclusion

We've only touched the tip of the proverbial iceberg with regard to SCM as a tool for testers, but it's clear that SCM isn't just for developers and release managers anymore. As testers, if we take advantage of the information that can be found in our SCM systems and use it to help guide our testing efforts at each phase of the development process, we'll not only be more effective at finding defects, but we may also find that our development colleagues will come to see us in a somewhat differ-

```

$ p4 describe -s 776

Change 776 by ines@ines-rose on 2005/01/27 08:58:34

  Add hooks to simplify debugging in ModuleA

Affected files ...

... //CompanyX/ProjectY/ModuleA/main_module.cpp#5 edit
... //CompanyX/ProjectY/ModuleA/main_module.h#5 edit
... //CompanyX/ProjectY/ModuleA/sub_module.cpp#16 edit

```

Figure 6

ent light. As we learn more about how the source code that implements our products is organized, which files are associated with which features, and which modules are more—or less—prone to defects, we may find that we are seen less as the bottleneck to getting a release out and more as valued contributors to the success of each new product or release. **{end}**

---

*William W. White has more than twenty years of experience in software application design and development, porting, technical writing, QA, and test. He currently manages the build/release and QA/test organizations at Perforce Software. In his spare time, Bill manages the Web site for the Alameda Floating Home Association and enjoys sailing in and around San Francisco Bay.*