Presentation
Paper
Bio
Return to Main Menu

P R E S E N T A T I O N

W3

Wednesday, February 14, 2001
10:15AM

BETTER TESTING — WORSE QUALITY?

Elisabeth Hendrickson
Aveo Inc

International Conference On
Software Management & Applications of Software Measurement
February 12-16, 2001
San Diego, CA, USA

Better Testing, Worse Quality?

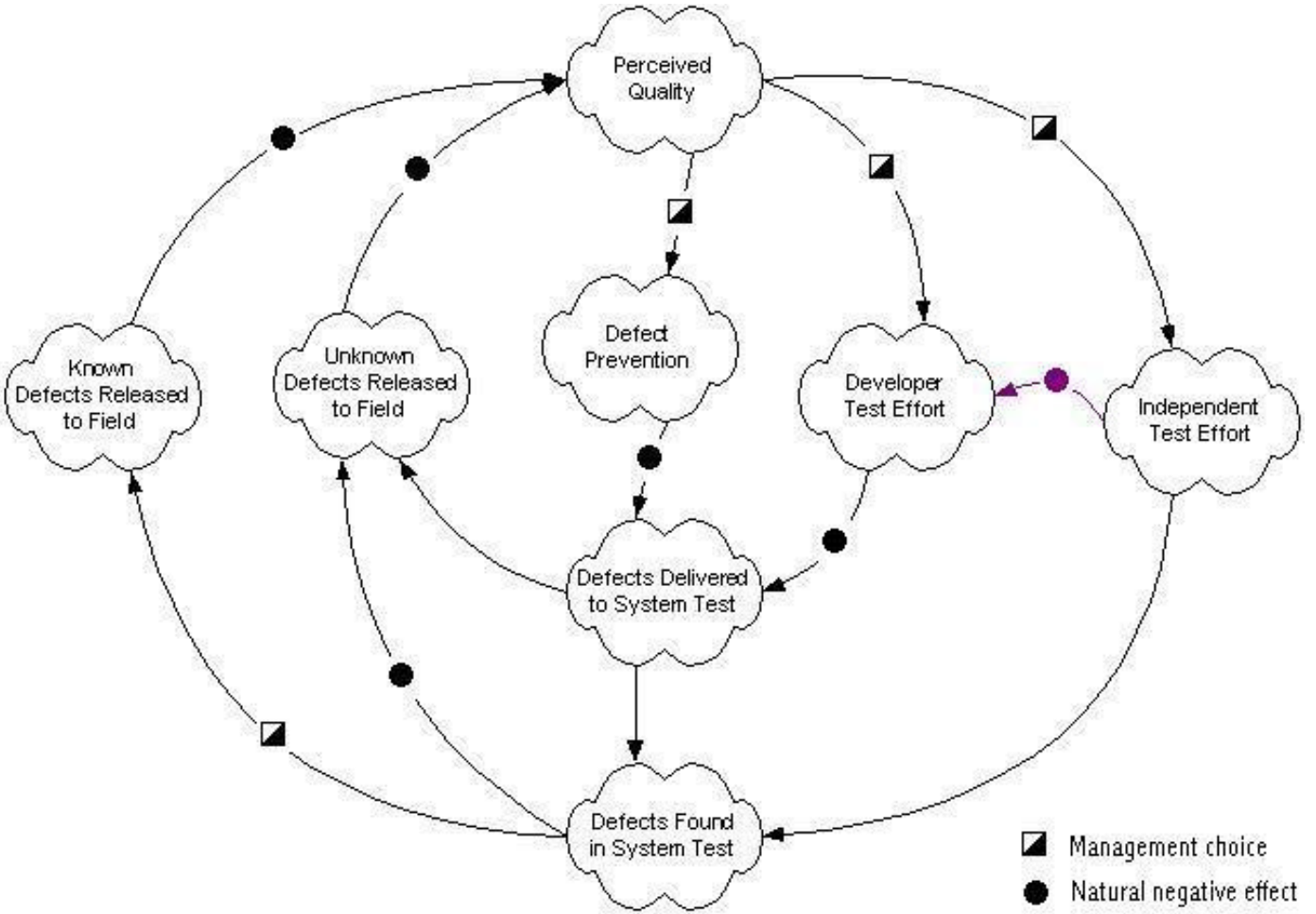
Elisabeth Hendrickson

SM 2001

Introducing Chuck, Our Hero

Chuck is a test manager at a software company. His boss, Susan, wants to know why her large investment in testing hasn't paid off in improved quality.

What Happened?



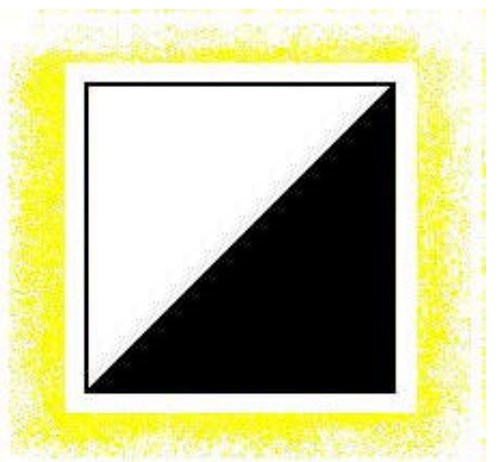
How Do You Know Your Software Is in a Death Spiral?

- Look at the bug counts
- Listen to the testers
- Watch integration times
- Listen to the developers
- Pay attention to resource demands

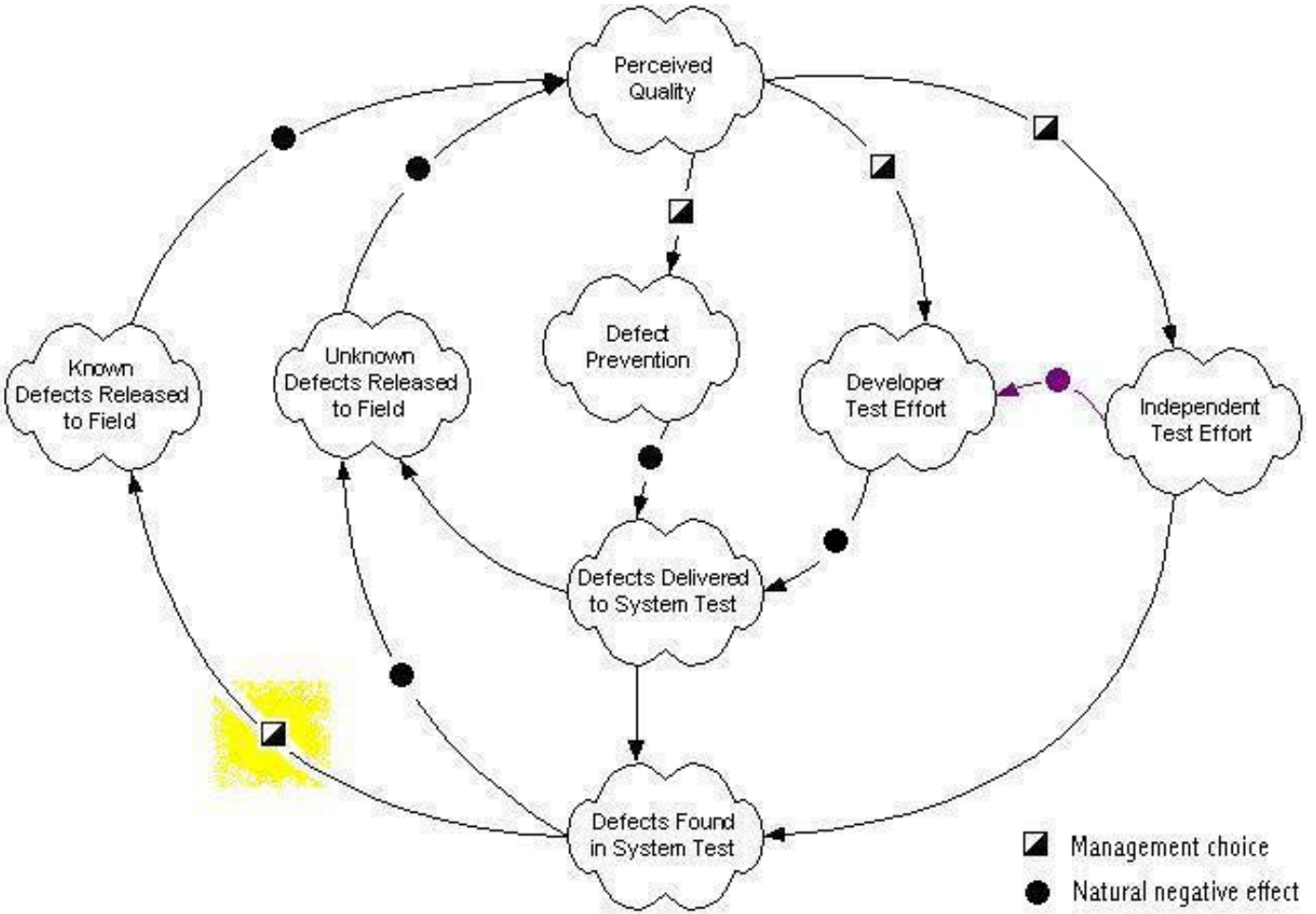
What Do You Do About It?

Remember the Other Control Points:

- Known Defects Released to Field
- Defect Prevention
- Developer Test Effort



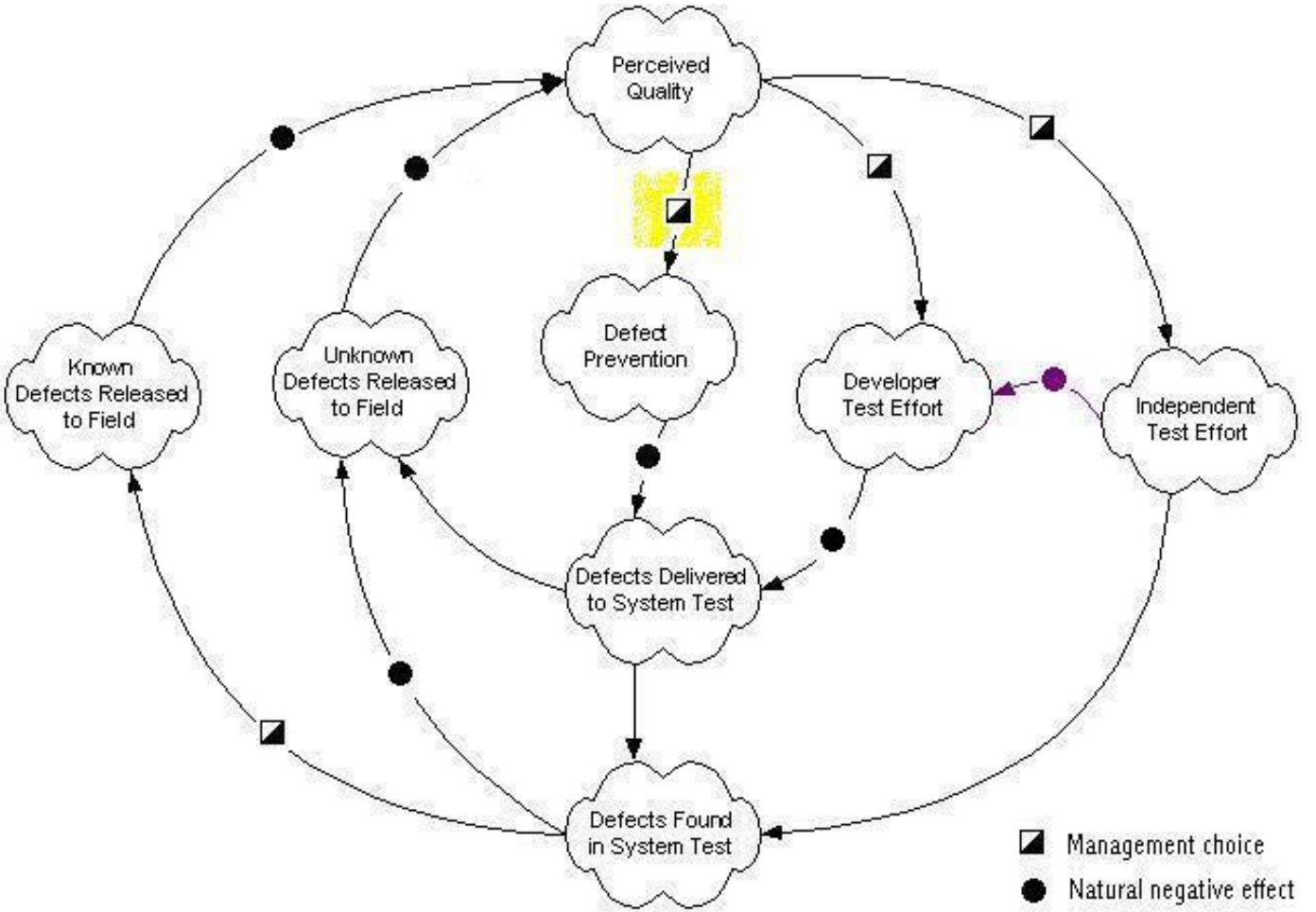
Known Defects Released to Field



Evaluate Known Bugs

- Can the developers keep up with the bug find rate?
- Are the testers finding the most important bugs?
- Is management making good decisions about the bugs being found?
- Is bug information used to improve the development process?

Defect Prevention



Assess the Requirements

- Make sure that everyone understands what they are building and why.
- Make implicit requirements explicit.
- Let the stakeholders see the software periodically throughout development.
- Listen carefully and question assumptions.

Bug Proof the Implementation

- Define coding standards and good coding practices.
- Hold reviews to make sure the coding standards are followed.
- Take advantage of code reviews as opportunities to share tips and techniques for bug proofing in your environment.

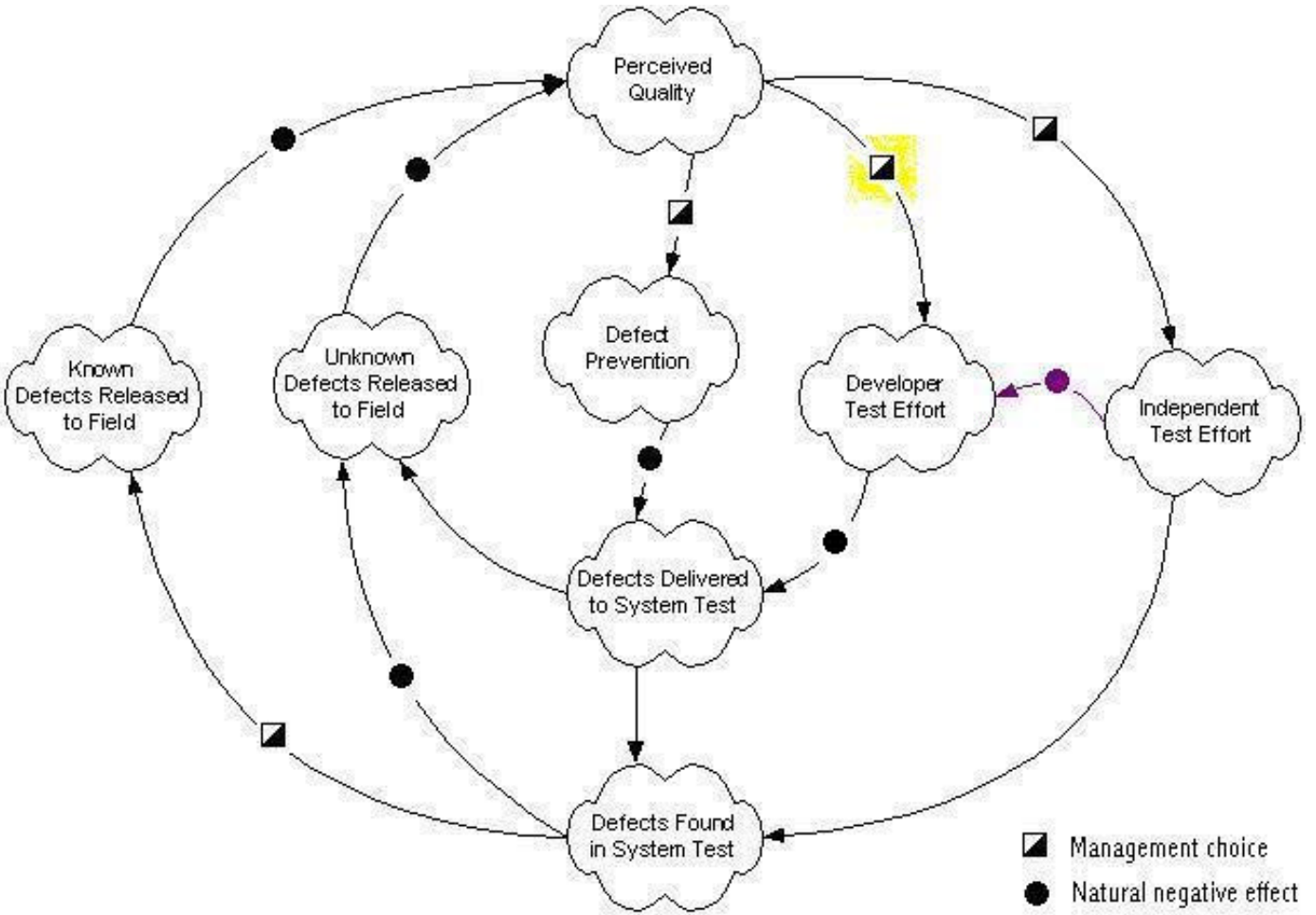
Don’t Let It Be Someone Else’s Problem

Watch out for a culture that encourages the idea:

Don’t Do Today What You Can Push Off Onto Someone Else’s Plate

If it doesn’t become someone else’s problem, it’s no one’s problem. The software will suffer as a result.

Developer Test Effort



Find Bugs Earlier

- How is "unit testing" defined?
- Who is responsible for unit testing?
- Can parts of the system be tested in isolation?
- Is the unit test effort improving?
- What tools are used to help unit testing?
- Is the system built and integrated daily?

Schedule Time for Developer Testing and Bug Fixing

- Include time for developers to run memory leak tools or other developer-oriented testing tools during development.
- Include time for developer to fix the bugs they find while testing their own code.
- Listen for signs of developers under time pressure to skip steps.

Better Testing, Better Quality?

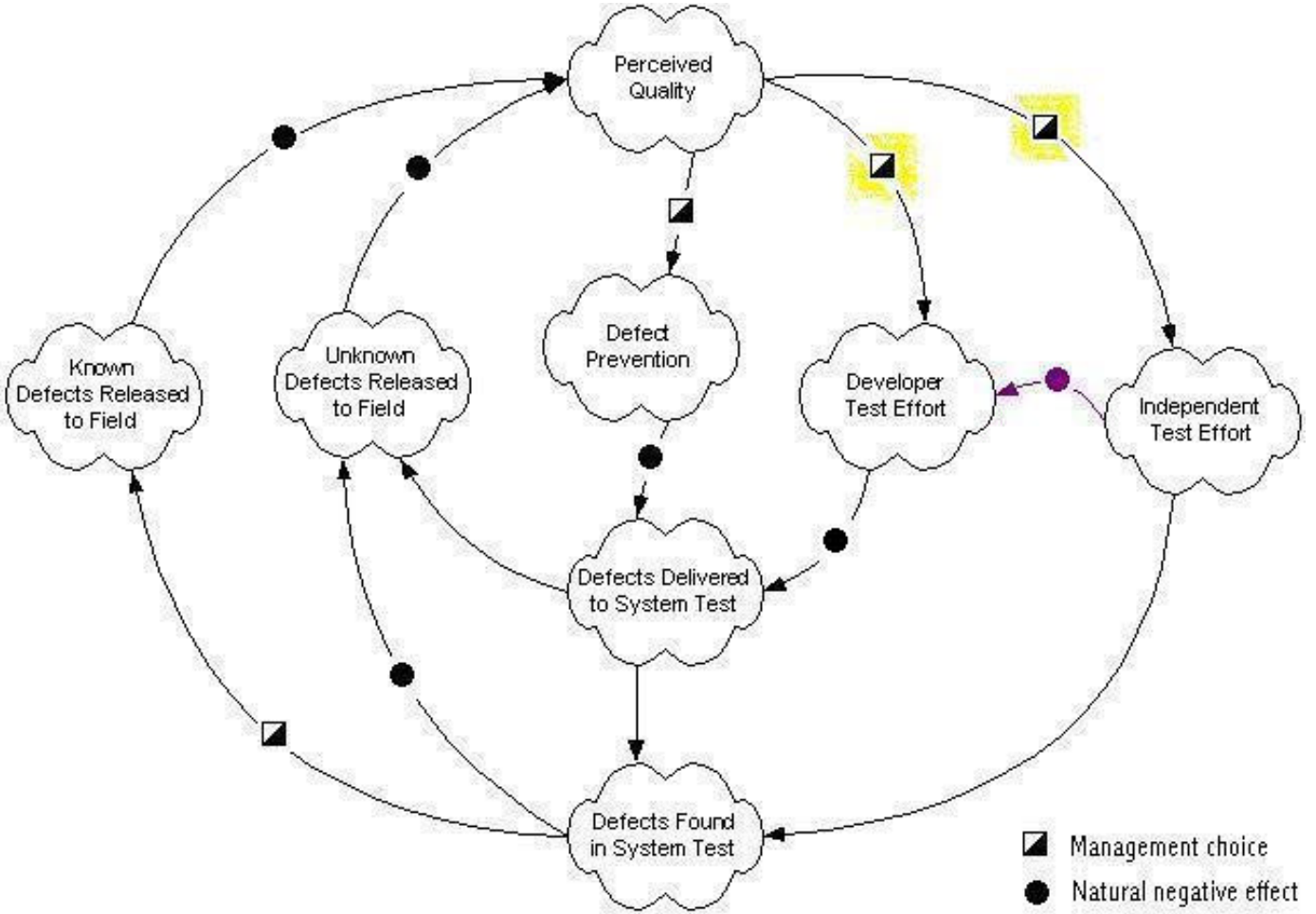
- Invest heavily in bug prevention and very early defect detection.
- Don’t plan to get better quality by only improving the independent test effort.
- Remember the other points of control: Defect Prevention; Developer Test Effort; and Known Bugs to Field.

Epilogue to Chuck's Story

So what happened to Chuck?

- He redeployed his staff so they appeared bottlenecked.
- Developers reacted—some were unhappy.
- The development lead insisted on extra developer testing.
- The quality of the code into test improved.

Why Did Chuck's Strategy Work?



References and Acknowledgements

- Weinberg, Gerald. *Quality Software Management: Volume 1: Systems Thinking*.
- Weinberg, Gerald. *Quality Software Management: Volume 2: First-Order Measurement*.
- Thielen, David. *No Bugs! Delivering Error - Free Code in C and C++*.

- Many thanks to Esther Derby, Nynke Fokma, Kirk Hendrickson, Cem Kaner, Nathaniel Lee, Pat McGee, Maureen O'Hara, Neal Reizer, and Johanna Rothman, for reviewing early drafts of this work and for their insights on system effects in software development.

Elisabeth Hendrickson
esh@qualitytree.com
http://www.qualitytree.com

Better Testing, Worse Quality?

December, 2000

If you are lucky, you've never been on the receiving end of a Vice President discovering that a large investment in testing hasn't paid off in improved quality.

A test manager at a company is about to get unlucky.

Chuck is the test manager for a small startup software company. He has a lot of experience setting up and running test groups. Susan is the Vice President to whom Chuck reports.

Chuck stood in the doorway to Susan's office. "You wanted to see me?"

"Yes. I just got out of the executive staff meeting." Susan motioned Chuck to come in and sit down. "We were discussing the recent increase in customer complaints."

Chuck took a seat across the desk from Susan, "I'd heard about that. As you can imagine, I'm not too pleased."

"Neither are the other executives; they had a few pointed questions for me about how this happened. So now I need answers. We've given you a well-stocked lab, you've hired a large team of experienced professionals, you've brought in training for them, and you've established good test practices. With all this investment in testing, how is it that our software is *worse*?"

How Did It Happen?

So what happened to Chuck? How did he end up sitting across the desk from Susan explaining why all the work he'd done to improve the testing had not paid off?

Before Chuck arrived, the testing had been done by a handful of largely untrained testers. "If you can't adequately test the software with the team you have," Susan had directed Chuck, "hire more testers!"

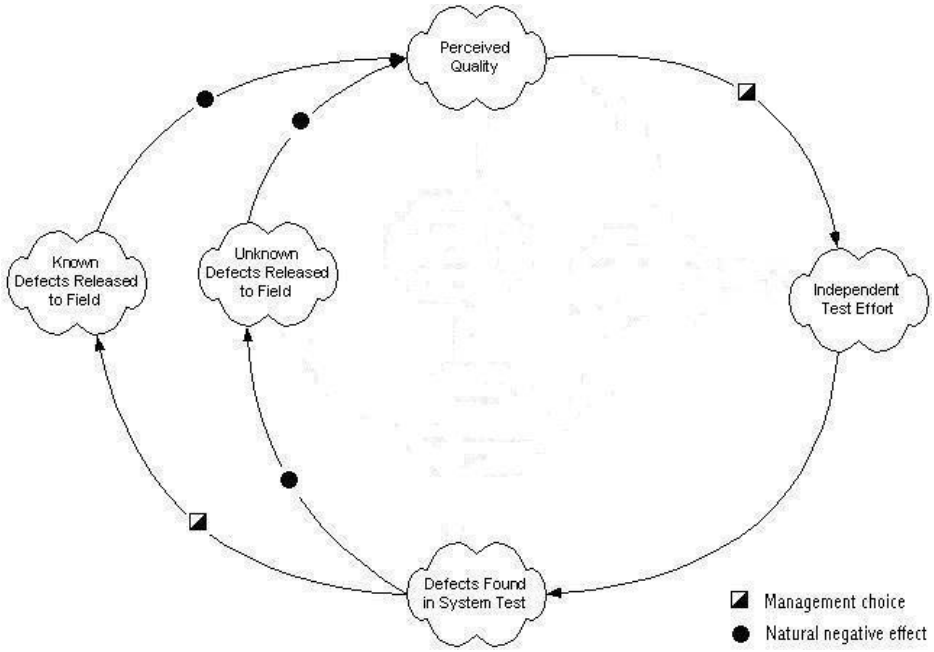
Chuck hired more testers. He quadrupled the size of the team. He established better test practices, documented the tests, and trained his team. The developers were ecstatic to see the test team growing by leaps and bounds. "This is great! They'll find all the bugs," they said, and merrily continued coding. The death spiral had begun.

The Software Death Spiral

Chuck slumped in his seat as Susan asked the question. "I know. I've been trying to figure this out myself ever since I heard about the problems in the field."

"Did we just miss the bugs?" Susan asked.

"I don't think it's quite that simple. Let me show you what I think happened." Chuck drew a diagram on the board as he spoke, "When I got here, you were concerned that the software was not of high enough quality. You put a lot of emphasis on defect detection—you told me to staff up the test team, establish a lab, design better tests."



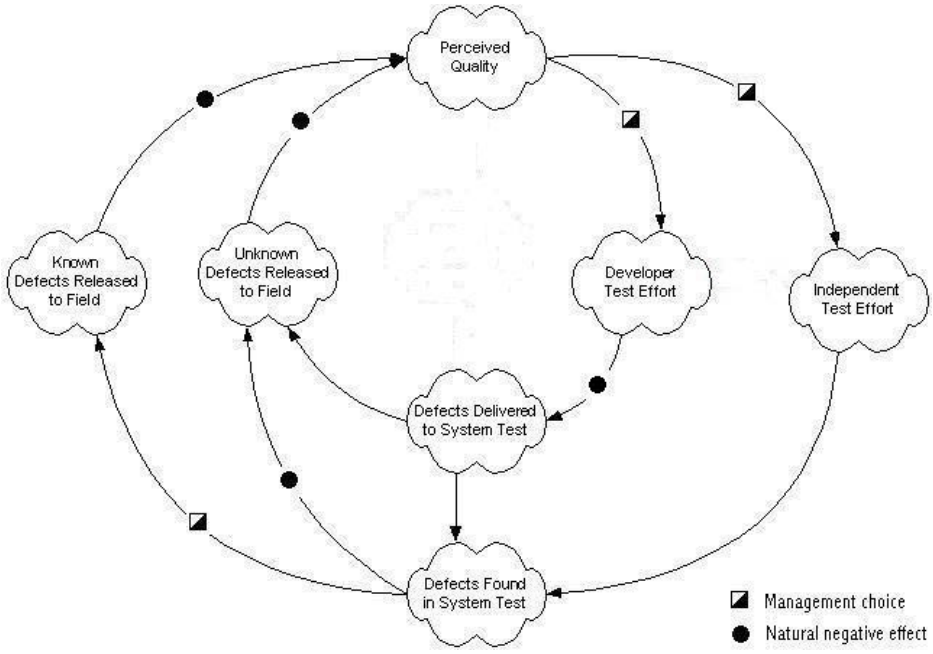
Chuck continued, "This diagram shows what you expected to have happen. You were making a choice to increase the independent test effort. The ◪ symbol indicates that you made a choice. You expected the decision to increase the independent test effort to result in an increase in the number of defects found in system test. And it did. We found more bugs in the last release than in the previous two releases combined."

Susan frowned at the unfamiliar symbols in the diagram. "So the symbol on the line between defects found in system test and defects released to the field means that we could choose whether or not to fix the bugs?"

"Yup," Chuck replied. "Fix the bugs, fewer known defects would ship to the field, and bingo—quality goes up. The ● symbol shows an inverse relationship between the known defects in the field and the perceived quality. As the number of bugs shipped to the field goes down, quality goes up, and vice versa. Similarly, as the number of defects found in system test goes up, the number of unknown bugs released to the field goes down."

"So why didn't it work?" Susan demanded.

Chuck turned back to the board, "I've been struggling with that question and I think I finally have an answer. Before I got here, the developers were doing a lot of their own testing. If we add them into this diagram, it looks like this:"



Chuck continued, "The more the developers tested, the fewer bugs were delivered to system test, and the fewer unknown bugs went to the field."

Susan tapped her foot in agitation. "OK, so the more bugs there are, the more escape to the field undetected. I get that."

"So what happens to the developer test effort when the independent test effort increases?" Chuck asked.

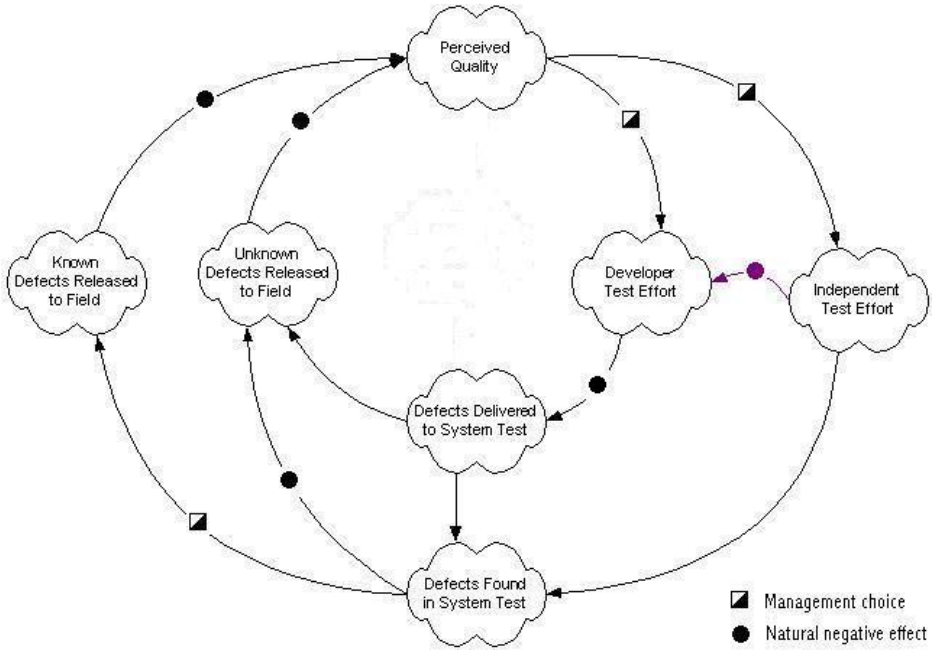
Susan stared at the board for a moment. "There's no connection on the diagram."

"Not yet. But do you think there should be? I realized last night that I think there is." Chuck responded.

"Oh."

"Oh?"

Susan sighed. "I just remembered a conversation I overheard recently. A developer was telling her lead that she wanted to spend a little more time making sure she got everything right. The lead replied, '*We're late already! Just get it into test!*'" Susan stood and reached for a whiteboard pen. "At the time, I thought that was the right answer. I wanted the software in test as well," she said as she drew an inverse relationship between Independent Test Effort and Developer Test Effort.



"Yup, I saw the same thing last night when I was muddling through the problem. It's an unanticipated side effect." Chuck replied. "The developers here thought that the independent test group had assumed full responsibility for all the testing—including the testing they used to do."

Susan focused back on Chuck, "Yes, I think you're right. And as a result, the number of defects delivered to test continued to go up."

"Attitudes made it worse: every time we found a show-stopper bug, we caught grief for it," Chuck shook his head. "But we continued catching them and schedules slipped. Even so, we couldn't catch all the bugs. It took longer to make worse software."

Susan shrugged. "So what do we do? If more testing will make the software worse, do we stop testing?"

Chuck exclaimed, "Goodness no! We have to test. Otherwise we can't be sure we met our quality goals before we ship. But we need to keep the developers from assuming that the test team is here to catch everything."

Susan continued staring at the picture, "And we can do that by…"

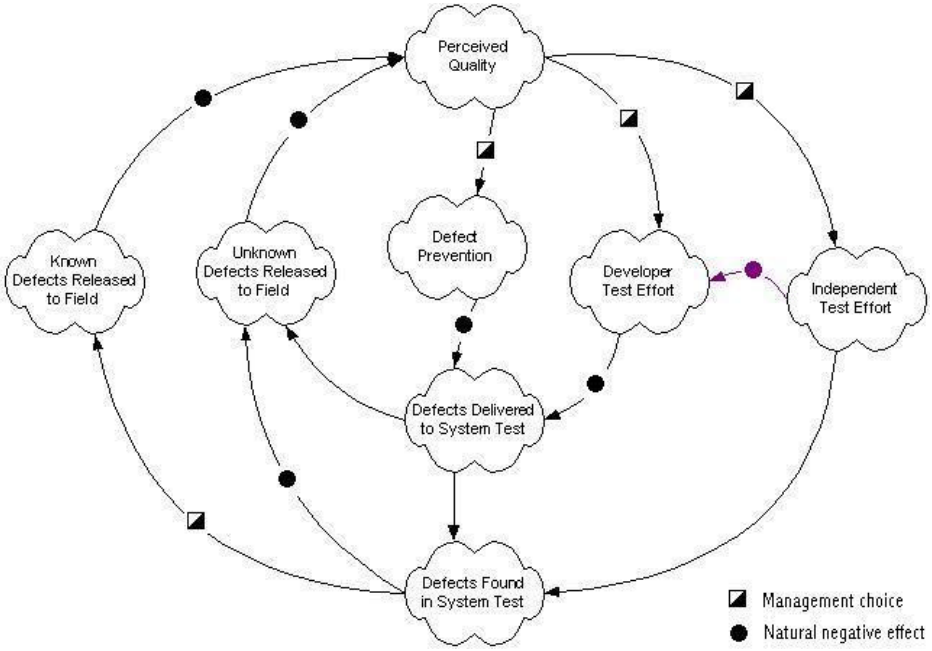
"Two ways. One is that you can make a choice to put more emphasis on developer testing. Two, we make it look like there's less testing resources available. Spread the folks we have now a little more thinly. There are kinds of tests we're not doing now that we really need to do. By taking some of the testers already on staff to do that work, we'd effectively be reducing the test effort—at least from the developers' perspective."

Susan asked, "What about the idea that having a developer test his own stuff is like the fox watching the henhouse?"

Chuck replied, "That's one of the great fallacies. Just because you have an independent tester evaluate the software before it ships doesn't mean that the person who created it should never even think of running a test on it. Like my high school math teacher always said ten minutes before the end of a test, 'Check your work!'"

Susan frowned at the diagram. "What if we put more emphasis on preventing bugs in the first place? What effect would that have?"

"Now you're talking!" Chuck exclaimed, adding another cloud to the diagram:



"The more effort we put on defect prevention—better requirements gathering, better coding practices, providing training opportunities for the developers—the fewer bugs sneak into the code to begin with!"

Epilogue

Chuck reorganized his department, turning one large testing team into a collection of smaller, more specialized testing teams. The developers noticed the shift immediately.

One development lead wanted to know, "Hey, when are we going to get more testers on our project. You're bottlenecked in test!"

"We're not going to get more testers." Chuck replied. "It's a funny thing about testing—the worse the software is when it gets to us, the longer it takes to get out. I suggest you work on making it right the first time."

The development lead didn't like that answer and he let Chuck know it, both publicly and privately. But Chuck didn't change his stance. Soon after, the development lead sent this email to his staff:

> As you know, there's a large, growing backlog currently in test.
>
> Development MUST produce bug-free code to ensure that the testers can pass our work quickly to release.
>
> Therefore, PLEASE test every component THOROUGHLY and do what you can to make sure it's bulletproof BEFORE requesting a build. Just because a project is in the queue for test doesn't mean you should leave it as-is - PLEASE consider performing a few hours of testing to ensure that it is working as well as possible!
>
> Thanks for your careful testing!!!

The result? A remarkable decrease in the number of bugs delivered to test; more predictable schedules; and increased quality to the customer.

How Do You Know Your Software Is in a Death Spiral?

The death spiral occurs when management chooses to turn a control point in the wrong direction, inadvertently making the situation worse. It's like turning the thermostat the wrong way. You think, "It's still cool in here; I'd better turn the thermostat up more to make it warmer." Then you continue turning the thermostat in the same direction you were turning it before, causing the room to become cooler yet.

So if you are responsible for delivering software and you're concerned about the quality, how can you tell if it's in a death spiral of increasingly bad quality?

The most obvious sign of trouble is that the software gets worse from release to release. However, if you wait until the pattern is that obvious, it's difficult to correct and may have already started having a negative effect on the business.

You can get early warning signs of software being in a death spiral if you look for the following warning signs:

- **Look at the bug counts.** If the independent testers are finding more and more bugs, are they getting better and better at testing, are there more bugs to find, or both? Increasing bug find rates are always worth investigating.
- **Listen to the testers.** When there are more bugs in the software than the testers can catch, testers complain. "It's completely broken!" When asked for more information about a bug they may reply, "I can't spend more time on that bug. There are too many more to report!" This tells you that there really is a bottleneck in test—the bugs are appearing faster than the testers can catch them.
- **Watch integration times.** It should not take longer and longer to get the whole system working together each cycle. If it does, it may mean that the system needs to be integrated more frequently to catch interoperability problems earlier. It may also mean that the components that make up the system are becoming more fragile or buggy over time.
- **Listen to the developers.** Most developers want to feel proud of the software they deliver. Developers under pressure to "throw it over the wall" may begin

complaining that they don't have enough time to run all their tests or fix all the bugs they find in unit testing. Take these complaints seriously. The developers are not just being perfectionists—they know better than anyone what state the code is in and what needs to be fixed.

- **Pay attention to resource demands.** Watch for areas that seem to be demanding more and more resources. Whenever you discover a bottleneck, you've just received excellent information about the trouble points in your process. Look to the point in the process one step before the bottleneck. For example, if the test group desperately needs more testers, it may be that the developers aren't doing enough of their own testing. If the developers can't seem to deliver on time, perhaps it's because they don't know what they are supposed to build: the requirements are poorly defined.

And What Do You Do About It?

As Chuck and Susan discussed, there are several control points in the software development system. As a manager, you can choose to increase or decrease the emphasis on and time allotted for:

- Bug fixing
- Defect prevention
- Earlier defect detection through developer testing
- Independent testing

Evaluate the Known Bugs

The first instinct of many managers when customers complain about a bad release is to throw testing resources at the problem—or blame the testers for letting bad software ship.

Before blaming the testers, look at what the developers are doing with the bugs that have already been reported. You may discover that the testers are already finding more bugs than the developers can handle. Adding more testers will simply exacerbate the problem. If the testers are finding more bugs than the developers can handle, add more developers, not more testers.

Also make sure that the testers are finding the kinds of bugs you care about. If the testers are focused exclusively on finding minor cosmetic defects, you may feel that they are re-arranging deck chairs on the Titanic. *"Who cares about the deck chairs?!? We've got a whopping hole in our side here! Start bailing!"* If this is the case it probably means that the testers don't fully understand the system or the requirements.

Make sure management is making good decisions about the bugs that the testers found. Is management deferring serious issues? If so, finding out why the issues are being deferred is more important than improving the testers' ability to find more issues.

Finally, bugs provide invaluable information about both the software and the software development process. Are a lot of the bugs in one particular area of the software? Consider reviewing that area thoroughly. Are most of the bugs requirements problems

rather than implementation problems? Your requirements process needs improvement. Evaluate your known issues to understand where problems are occurring. Use that information to continuously improve the software and the process by which the software is created.

Defect Prevention

If you want to improve the quality of the software, stop the problem at the source. Begin by looking at the requirements: most bugs are caused by a lack of agreement on the requirements, not by simple developer mistakes. Also identify bug-proofing techniques to improve the quality of the implementation.

Assess the Requirements

Make sure that everyone understands what they are building and why. Poorly defined requirements usually result in rework. If there was barely enough time in the schedule to implement a feature once, there certainly isn't time to implement it twice—once the wrong way as interpreted from the requirements and once the way the stakeholders actually wanted it.

Also, if there are requirements that aren't being met, make sure they are explicit. Implicit requirements such as, "Must be consistent with our branding," may not be considered in designing and testing the software—the requirement is obvious to the people defining the requirements from a business perspective, but may not be obvious from a technical perspective.

Periodically show the stakeholders what you have done so far. This gives them an opportunity to comment on direction before it's too late. It would be a really bad thing for the marketing person to say two days before launch, "Oh, I thought it would be a Java applet inside a browser window, not a Windows application."

Bug Proof the Implementation

Simple errors in implementation can cause major problems in the software under test. Consider the following error:

```
if (myvalue = 0){
      Do Something Here
} else {
      Do Something Else
}
```

In this example, we want to see if the value of "myvalue" is 0. However, we're actually setting the value to 0 in all cases.

In many languages, the double equal signs ("==") means "compare" while a single equal sign ("=") means "set value". If the developer mistakenly uses "=" where she meant to use "==", the meaning of the code changes entirely. In this case, the "Do Something Else" code would never execute. Behold, a bug.

To prevent this particular bug, many developers will use "`0 == myvalue`". The compiler will produce an error if the developer mistakenly wrote "`0 = myvalue`" because 0 cannot be reset.

There are many other examples of techniques for bug proofing implementation. For more suggestions, see David Thielen's book, *No Bugs! Delivering Error - Free Code in C and C++*.

Techniques for bug proofing code vary depending on the development environment. Identify bug-proofing opportunities like this in your environment. Develop coding standards that reflect these bug proofing techniques and educate developers in how to avoid easy-to-prevent bugs. Hold code reviews to ensure that the standards are followed and to provide a forum for exchanging tips and tricks for bug proofing.

Don't Let It Be Someone Else's Problem

In an environment with tight deadlines and an emphasis on schedules (real or perceived), some people develop the maxim:

Don't Do Today What You Can Push Off Onto Someone Else's Plate

This can be an effective strategy for an individual dealing with far too much work and far too little time. However, it is probably not an effective strategy for the organization.

For example, a developer under a lot of time pressure may find herself thinking, "I don't have time to build a test harness to do my unit tests properly. But that's what the test department is for."

In reality, the test department may or may not be set up to create those harnesses. If the test department is not set up for that kind of testing, the developer has not succeeded in making unit testing someone else's problem. Now it's no one's problem—and the software will suffer as a result. Thinking that testing is someone else's problem is the key reason for the death spiral Chuck and Susan observed.

Developer Test Effort

If the independent testers first see the software a week before you plan to ship, it's probably too late for them to help improve the quality at all. Make sure the testers get involved earlier—preferably from the time the requirements are generated.

That's just one way to ensure that the software it tested earlier. It's even more important to look at the development process:

- How is "unit testing" defined and who is responsible for it?
- Are there custom test harnesses to allow parts of the system to be tested independently?
- How do you measure the efficacy of the unit test effort?
- How do you improve the unit test effort on an ongoing basis?
- Do you use memory leak tools, and if so, do you use pre-defined test scripts with the tools to ensure you touch on all the areas of the software?

- Do you build and integrate the system on a daily basis?

If the answer to any of these questions is "I don't know" or "I don't think we do," investigate a little more, then consider making some changes. The sooner a bug is found, the less it costs to fix.

Also remember to schedule time for developer testing and bug fixing. Developers often provide schedule estimates that only include activities necessary to get to code complete. The schedules don't call out activities such as building test harnesses or taking time to run tools such as *Purify*, *BoundsChecker*, or *lint*. If you are in a position to do so, insist that the development schedule include time for these kinds of activities.

Better Testing, Better Quality?

As Chuck and Susan found, investing heavily in an independent test effort is not the best way to increase the quality of the software. Remember the other management control points in the system diagram that Chuck and Susan drew:

- Defect Prevention
- Developer Test Effort
- Known Defects Released to Field

The key to improving the quality of your software is to invest heavily in bug prevention and very early defect detection. It's best if the bug never happens. But if it does, the closer to the original developer's desk the bug is found, the better.

If anything seems to be discouraging the developers from spending time on bug proofing and testing activities, find out why and eliminate all reasons for not spending time on these critical activities.

Finally, remember that testing is a way to measure quality; it is not a remedy by itself. At the very least, if you want to improve the quality of the software before release, you must commit to fix the bugs found in testing.

So is it possible to have better testing and better quality? Certainly—as long as you don't plan to get better quality by only improving the independent test effort.

Bibliography

Weinberg, Gerald. *Quality Software Management : Volume 1 : Systems Thinking*. Dorset House, 1991

Weinberg, Gerald. *Quality Software Management : Volume 2: First-Order Measurement*. Dorset House, 1993.

Thielen, David. *No Bugs! Delivering Error - Free Code in C and C++*. Addison Wesley Publishing Company, 1992.

Acknowledgements

Many thanks to Esther Derby, Nynke Fokma, Kirk Hendrickson, Cem Kaner, Nathaniel Lee, Pat McGee, Maureen O'Hara, Neal Reizer, and Johanna Rothman, for reviewing early drafts of this work and for their insights on system effects in software development.

Elisabeth Hendrickson

Elisabeth Hendrickson is currently an independent consultant specializing in software quality and testing. With over a dozen years of experience in the software field, Elisabeth has seen several examples of better testing leading to worse quality. Prior to becoming an independent consultant, Elisabeth was the Director of Quality Engineering and Project Management at Aveo Inc. You can read more of Elisabeth's thoughts on software management, quality, and testing at www.qualitytree.com or reach her at esh@qualitytree.com.