Software Testing:

A Profession of Paradoxes

By Zeger Van Hese

Testing Consultant,
CTG Belgium

Introduction

For many years now, testing professionals have struggled to define best practices, methodologies, standards and qualification schemes for optimal software testing. Much work has already been done in this area, and continues today. But will it ever be possible to fully explore and grasp all aspects of software testing?

It's certainly not an easy task, for this highly-regulated profession seems to harbor some remarkable irregularities that are embedded in its very nature. Paradoxes and 'catch-22s' abound throughout the various domains of software testing, including test execution, usability testing, test report data, test team, test automation, and developer testing.

This paper aims to create awareness of a number of testing paradoxes that challenge us all. Some are well-known while others are more obscure—but all are worth investigating. The paper also considers a few critical questions for software testers. How do we cope with situations that defy our intuition? Do we take them for granted, or is there a solution at hand? How can we can learn from them, anticipate them, and use them to get better at what we do?

A Special Report

© Computer Task Group, Inc. 2007 All rights reserved.

© Computer Task Group, Inc. 2007

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photography, recording, or any information storage and retrieval system, without permission in writing from CTG.

© Computer Task Group, Inc. 2007 All rights reserved.

Some Definitions

Catch-22

The term ‘catch-22’ was first used in the Joseph Heller novel *Catch-22* to describe a paradox in a law, regulation, or practice where one is a victim regardless of the choice he makes. In the novel, a U.S. Air Force pilot who wants to be excused from combat flight duty is required to supply an official medical diagnosis that he is unfit for duty because he is insane. However—and here’s the catch-22—since no sane person would want to fly dangerous combat missions, the pilot, by attempting to avoid them, demonstrates that he is in fact sane, and therefore fit to fly.

Over time, ‘catch-22’ has acquired a more universal meaning. Today, we define it as a ‘no-win’ dilemma where the only solution is to follow a rule that is blocked by another rule, or a paradoxical situation where you cannot obtain A without B, but B first requires A (this is also known as the ‘chicken/egg problem’). A classic example of a catch-22 occurs in the context of searching for a job. In moving from school to a career, job seekers often find they are unable to find a position without work experience—experience that can only be attained on the job.

Paradox

A ‘paradox’ is an apparently true statement or group of statements that leads to a contradiction or a situation that defies intuition. Paradoxes are frequently utilized to arrest attention and provoke fresh thought. The word ‘paradox’ is often used interchangeably (and wrongly) with contradiction; but whereas a contradiction asserts its own opposite, many paradoxes do allow for resolution of some kind, with the contradiction often disappearing when you look at the problem from a different perspective.

The word ‘paradox’ is often used interchangeably (and wrongly) with contradiction; but whereas a contradiction asserts its own opposite, many paradoxes do allow for resolution of some kind, with the contradiction often disappearing when you look at the problem from a different perspective.

The most famous paradox is probably Zeno’s paradox, involving a race between the Greek hero Achilles and a tortoise. The tortoise gets a head start, and apparently Achilles can never surpass it, since whenever Achilles runs to where the tortoise has been, the tortoise has moved further ahead. While Achilles makes up that gap, the tortoise creates a new one, and so on. As long as Achilles’ forward progress is defined in terms of the turtle’s progress, the paradox holds, but as soon as we look at it from a different perspective—one that views Achilles and the tortoise both progressing toward the finish line—the paradox disappears.[1]

Paradoxes exist in most scientific domains: logical, mathematical, philosophical, and physical, and examination of the ambiguities, equivocations, and unstated assumptions that underlie them has led to significant advances in science, philosophy, and mathematics. [2]

© Computer Task Group, Inc. 2007 All rights reserved.

Testing is merely stopped; it can never truly be finished. Yet, at some point, we have to end testing and ship the software. The question is, when?

Equivocation

An 'equivocation' is the misleading use of a word with more than one meaning by glossing over which meaning is intended at a particular time. Using an equivocation, we could 'prove' that a ham sandwich is better than happiness[3] as follows:

> *Which is better, eternal happiness or a ham sandwich?*
>
> *It would appear that eternal happiness is better, but think again!*
>
> *After all, nothing is better than eternal happiness,*
>
> *and a ham sandwich is certainly better than nothing.*
>
> *Therefore a ham sandwich is better than eternal happiness.*
>
> —*Raymond Smullyan*

The Catch-22 of Testing

The complexity of software in general is the main cause of the catch-22 of software testing: testing is potentially endless. Even a program of moderate complexity can never be completely tested, since it is impossible to test until all the defects are unearthed and removed. As a result, software that is a hundred percent bug-free, even though thoroughly tested, does not exist.

Testing is merely stopped; it can never truly be finished. Yet, at some point, we have to end testing and ship the software. The question is, when? This is one of the most difficult issues to resolve in the field of software testing. On one hand, marketing a program with annoying faults undercuts your reputation and credibility. On the other hand, if you never start selling your product, you will soon go out of business![4]

One rule of thumb is that testing should continue as long as bugs are being found fairly regularly. Other indicators that can help determine when to stop include:

- Deadlines (e.g., release or test deadlines) are reached.
- Test cases are completed with a certain percentage passed.
- Coverage of code, functionality, or requirements reaches a specified point.
- The bug rate falls below a certain level.
- The test budget has been depleted.
- The risk associated with the project is below an acceptable limit.

Realistically, testing is a tradeoff among priorities—budget, time, and quality—that is driven by profit models. The pragmatic, and unfortunately most common, approach is to stop testing whenever some or any of the allocated resources (time, budget, or test cases) are exhausted. The optimal 'stopping rule' is to end testing either when reliability meets the requirement, or when the benefit to be derived from continuing testing cannot justify the testing cost.[5]

© Computer Task Group, Inc. 2007 All rights reserved.

Paradoxes in Software Testing

Testing Realities

Some particular situations illustrate what I'll call 'realities' of software testing. They're not really paradoxes, but rather axioms, truisms, or even aphorisms about the nature of software testing itself, although some are indeed paradoxical in nature. It is interesting to take a closer look at these realities, because each one of them offers a bit of knowledge that can help to put some aspects of the overall software testing process into perspective.

The first testing truism questions the role of the testers as gatekeepers of quality: who watches the watchmen?

Who watches the watchmen?

The first testing truism questions the role of the testers as gatekeepers of quality: who watches the watchmen? As Sara Ford states: "I was most surprised the day I realized the paradox of—how am I going to write tests for the tests that I'm writing?" [6]

One of the trickiest axioms within testing is that everyone relies on testers to find others' errors, while tending to forget that testers may be wrong themselves. Humans err, and testers are human. This means they overlook problems or misunderstand outputs. Some of the problems they find turn out not to be problems at all. Moreover, human memory is imperfect, so testers who rely too heavily on memory are bound to make mistakes. For that reason, it's important to take notes during testing: log what you did and what the system did. Notes are always more trustworthy than memory![7]

Another phenomenon is 'inattentional blindness', where an observer whose mind is otherwise occupied may fail to see things in plain view. When presented with the same scenario a second time, the observer may detect 'new' things and assume they weren't there before. Inattentional blindness means that unless we pay close attention, we can miss even the most conspicuous events that occur while we're executing our well-planned tests. This in turn suggests that perhaps our tests are not as powerful as we think.[8]

Joel Spolsky provides a good example of this principle:

> *All the testing we did, meticulously pulling down every menu and seeing if it worked right, didn't uncover the showstoppers that made it impossible to do what the product was intended to allow. Trying to use the product, as a customer would, found these showstoppers in a minute.* [9]

In 2005, Matthew Heusser identified some other classic testing mistakes: [10]

Test management mistakes:

- Dehumanizing the test process by relying too heavily on spreadsheets or MS project plans and disregarding the individual (also known as 'management by spreadsheet/MS Project')

© Computer Task Group, Inc. 2007 All rights reserved.

- Assuming that only testers are responsible for quality; blaming individual testers for letting bugs slip through
- Evaluating testers by bugs found, possibly resulting in a testing focus on trivial and easy-to-find bugs

Test automation mistakes:

- Adding test automation when testing is late, disregarding the fact that somebody has to learn the tool and record the scripts
- Assuming that the software is bug-free just because nothing was found during an automated test run
- Assuming that hiring test tool skills is sufficient; failing to realize that test tool skills can be taught, but talent cannot

It is an inherent quality of software testing that it can show that bugs exist, but not that bugs don't exist.

Test strategy mistakes:

- Including insufficient diversity in the test strategy (only requirements-based testing, for example), and as a consequence, missing entire classifications of defects
- Relying too much on scripted testing; not thinking enough as an end-user of the product (as mentioned earlier, inattentional blindness may cause us to overlook even the most apparent problems)
- Using untrained exploratory testers, which fails to recognize that exploratory testing is a discipline in its own right (not every tester is a good exploratory tester by nature: it requires specific skills)
- Placing too much emphasis on documentation (time spent documenting is time not spent testing)

Testing can't show that bugs don't exist

If testing fails to find bugs, no matter how hard we try, does this mean that the software was cleanly written and that there are indeed few if any bugs to be found? Not necessarily. It is an inherent quality of software testing that it can show that bugs exist, but not that bugs *don't* exist. You can perform your tests and find and report bugs, but you can never guarantee that there are no more bugs to find. You can only continue your testing and possibly find more.

Ron Patton (2000) compares this situation to an exterminator charged with examining a house for bugs. He inspects one house and finds evidence of bugs—live bugs, dead bugs, or nests. He can safely say that the house has bugs. He then visits another house, where he finds no apparent evidence of bugs. He looks in all the obvious places and sees no signs of infestation. Can he absolutely, positively state that the house is bug-free? No. All he can conclude is that his search failed to detect any live bugs. Unless he completely tears the house down, he can't be sure that he didn't simply miss them.[11]

© Computer Task Group, Inc. 2007 All rights reserved.

The very act of testing influences its outcome

This axiom is also often referred to as the ‘observer effect’. The idea is that since any form of observation is also an interaction, the act of testing itself can also affect that which is being tested. For example:

- When log files are used in testing to record progress or events, the application under test may slow down drastically.
- The act of viewing log files while a piece of software is running can cause an I/O error, which may cause it to stop.
- Observing the performance of a CPU by running both the observed and observing programs on the same machine will lead to inaccurate results because the observer program itself affects the CPU performance.
- Observing (debugging) a running program by modifying its source code (e.g., adding extra output or generating log files) or by running it in a debugger may cause certain bugs to diminish or change their behavior, creating extra difficulty for the person trying to isolate the bug (also known as a ‘Heisenbug’)

Nobody likes to make mistakes and everyone fears failure to some degree, but in retrospect, most breakthroughs depend on it.

Paradoxically, software testing is not always considered as the best way toward better quality. As Boris Beizer asserts in his complexity barrier principle[12], chances are that testing and fixing problems may not necessarily improve the quality and reliability of the software. Sometimes fixing a problem may introduce much more severe problems into the system.

Failure breeds success

It may be a cliche to say that we learn more from our mistakes than from our successes, but now there is scientific proof. Psychologists from the University of Exeter have actually identified an early warning signal in the brain that helps prevent the repetition of previous mistakes. Published in the *Journal of Cognitive Neuroscience*, their research identifies, for the first time, a mechanism in the brain that reacts—in just 0.1 seconds—to stimuli that have led to errors in the past. By monitoring activity in the brain as it occurs, the researchers were able to identify the moment at which this mechanism kicks in.[13]

Nobody likes to make mistakes and everyone fears failure to some degree, but in retrospect, most breakthroughs depend on it. Even the most successful companies have embraced their mistakes and learned from them. The truism ‘failure breeds success’ also applies to software testing. Here are some ways that failure can, in fact, be more beneficial for us than success:

- Failure encourages lateral thinking. It encourages us to look for other solutions that we might not have thought of had things been easier.
- Making mistakes makes us more experienced. As we err and learn from our mistakes, we tend to become more aware of what needs to be done in order to achieve our objectives.
- Mistakes build character by making us more thick-skinned: an extra asset for a tester, who needs to cultivate a critical attitude.

© Computer Task Group, Inc. 2007 All rights reserved.

- Failure forces you to be honest with yourself. If you keep getting disappointing results, maybe you'll eventually decide that testing is not your profession. If not, you'll know what you want even more clearly.
- Too much success too soon can lead to a false sense of confidence. The risk is that we become comfortable and lazy, losing our ability to self-critique our performance and thus some core testing qualities.
- Failure encourages improvement and planning. If the same mistakes keep reappearing, it's time to sit down and re-analyze your approach. Failing can be a way of finding out that the methods you used probably weren't the right ones. After reorganizing and planning, you should be better off than before.

Many observers have pointed out the similarities between real bugs and software bugs. Both types are social creatures that tend to stick together. Bugs follow bugs: if you encounter a defect, chances are that there will be more nearby.

The biggest bugs are the hardest to find

Once in a while we stumble on defects that are very hard to reproduce. These issues are called 'intermittent bugs'. An intermittent bug is the mysterious and undesirable behavior of a system, observed at least once, that we cannot yet manifest on demand.[14] The situation becomes even more difficult if these problems introduce system crashes. Upon restarting the system, any corrupted data in the memory may be deleted, thus destroying the evidence. In many cases, intermittent bugs are the result of long-term corruption of some resource or memory.

Good examples of this are memory leaks. Some function in the program does not return unused memory when finishing. Because there is a lot of available memory, this can go on for a long time, until the memory is depleted. The problem becomes even more complicated if this does not happen every time, but only in very special situations. If intermittent bugs occur, it's a good idea to rerun the same sequence of test cases, maybe even with the same timing, and do more checking than before.[15]

While intermittent bugs can mean serious trouble, they also provide an opportunity for us to do our best. The ability and confidence to investigate an intermittent bug are qualities that mark an excellent tester. The challenge is to transform the intermittent bug into a 'regular' bug by resolving the mystery surrounding it.

The more bugs you find, the more bugs there are

Many observers have pointed out the similarities between real bugs and software bugs. Both types are social creatures that tend to stick together. Bugs follow bugs: if you encounter a defect, chances are that there will be more nearby. Frequently, a tester will go for long spells without finding a bug. Then, he'll find one bug, then quickly another, and then another. This is not surprising, and in fact, there are several reasons why this is so (Ron Patton):

- Programmers have bad days. Code written one day may be perfect; code written another may be sloppy. One bug can be a tell-tale sign that there are more nearby.
- Programmers have habits. A programmer who is prone to a certain error will often repeat it.
- Some bugs are really just the tip of the iceberg. Very often, the software's design or architecture has a fundamental problem. A tester may find several bugs that at first seem unrelated, but eventually are discovered to have one primary, serious cause.

© Computer Task Group, Inc. 2007 All rights reserved.

The Pesticide Paradox

> *Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual.*[16]
>
> —Boris Beizer

When we have a test that removes one or more errors, running that same test over and over again will not eliminate errors that were previously removed, so the test becomes ineffective.

In 1990, Boris Beizer described the 'pesticide paradox' phenomenon: the more you test software, the more immune it becomes to your tests. He asserted that software undergoing the same repetitive tests eventually builds up resistance to them, similar to the reaction of insects to pesticides: if you keep applying the same pesticide, the insects eventually build up resistance and the pesticide no longer works.

When we have a test that removes one or more errors, running that same test over and over again will not eliminate errors that were previously removed, so the test becomes ineffective. Related to this, errors that remain get harder to detect. After several iterations, all the bugs that those tests would find have been exposed. Continuing to run them won't reveal anything new.

> *Highly repeatable testing can actually minimize the chance of discovering all the important problems, for the same reason that stepping in someone else's footprints minimizes the chance of being blown up by a land mine.* [17]
>
> —James Bach

The phenomenon of product code becoming more resistant to the test code, resulting in fewer bugs being found over time, is also a major drawback of traditional automated testing. When you design an automated test (equivalent to manually running a test case for the first time), you'll either find a bug in the functionality being tested, or the functionality will work as expected. Most likely, if you do not find a bug at this point, your tests will not find another bug in their lifetimes.

Tests of this kind do serve one important purpose, however: that of regression testing—that is, the testing of new code for side effects that break existing functionality. Regression testing is even more important—if not essential—when testing in agile development environments, where continuous automated regression is a must.

To overcome the pesticide paradox, software testers must be willing to continually design new test cases that cover all or most of the scenarios where bugs might be present. They might also adopt a methodology that allows them to reuse the test code to automatically test new scenarios and code paths (better known as model-based testing). In model-based testing, a machine creates and runs the new test cases and validates behavior of the system, with elimination of all manual intervention.[18]

© Computer Task Group, Inc. 2007 All rights reserved.

Mathematical Paradoxes

This section illustrates two interesting cases that can occur when performing software testing. They're fundamentally mathematical in nature, but they can be a useful addition to our standard troubleshooting toolbox.[19]

Simpson's Paradox (or a possible test reporting paradox)

'Simpson's Paradox' is a situation in which two or more sets of data lead to one conclusion when evaluated individually, but lead to an opposite conclusion when the sets are combined. In software testing, this is a situation that can occur when software system B is worse in every area of comparison than software system A (Figure 1), yet a combined test report indicates that system B is the better system (Figure 2).

'Simpson's Paradox' is a situation in which two or more sets of data lead to one conclusion when evaluated individually, but lead to an opposite conclusion when the sets are combined.

Figure 1 Comparing System A and System B

Manual Tests	System A	System B
Number of tests passed	50	15
Total number of tests	200	100
% tests passed	25%	15%
Automated Regression Tests	**System A**	**System B**
Number of tests passed	85	300
Total number of tests	100	400
% tests passed	85%	75%

Figure 2 Combined Data for Manual and Automated Regression Tests

Combined Data	System A	System B
Number of tests passed	135	315
Total number of tests	300	500
Score	45%	63%

The moral is that when reviewing test result data, you should first question whether the data is aggregated from other sources. If it is, you need to look at the original uncombined data. Additionally, when generating a test result data report, be very careful about combining the data in an attempt to simplify your presentation. When people say that "statistics can be made to say whatever you want," this is exactly the sort of situation they're referring to.

Simpson's Paradox occurs quite frequently, and can lead to erroneous test conclusions that can, in turn, prompt the wrong business decisions. The paradox can easily sneak by us, especially if we are presented with summary data only, and don't get a chance to see the original, uncombined data.

© Computer Task Group, Inc. 2007 All rights reserved.

Braess's Paradox

Consider these two facts:

- When 42nd street was closed in New York City, instead of the predicted traffic gridlock, traffic flow actually improved.
- When a new road was constructed in Stuttgart, Germany, traffic flow worsened, and only improved after the road was torn up.

These paradoxical phenomena are two real-world examples of the Braess paradox, named after Dietrich Braess who, in 1968, noted that adding extra capacity to a network where the moving entities selfishly choose their route can in some cases reduce overall performance. Braess's Paradox has been investigated heavily by researchers. Because of its obvious relation to data packets traveling on a network system, we can informally rephrase the paradox as: "It is sometimes possible to increase network congestion by increasing the number of routes between nodes".

Adding a perfectly good, load-balancing server can actually reduce network performance in a most surprising way.

From a software testing point of view, Braess's Paradox can arise when running network performance tests. Adding a perfectly good, load-balancing server can actually reduce network performance in a most surprising way. While the chances of encountering Braess's Paradox are very slim, the phenomenon does exist. The moral is that you should not assume that adding capacity to a network will necessarily improve performance. If you add capacity and do not see the performance improvement you were expecting, Braess's Paradox is one of the things to investigate.

Test Team Paradox

In 2006, William Echlin warned of the danger that affects every test team: good software testers need to be continually critical of other people's work, but this attitude of continuous criticism can affect other aspects of our work. A tester has to look for all the negative aspects of the software. The problem for many testers is that this pessimistic, critical attitude can easily come to extend beyond the software they are testing.

This is where the paradox lies. If your test team is positive and happy, you may have the wrong people for the job. A successful team needs to be critical; judgmental; even negative. But a successful team should also try not to jeopardize relations with other teams and within the team itself. The key for success as a team is to maintain a critical focus while having a positive outlook. These actions can help to achieve that goal:

- Highlight the good points within your team or project. It's easy to maintain a critical attitude, but it shouldn't be too hard to come up with positive comments about the team, the project, or the test process.
- Set realistic, clear, and achievable goals for both the team and yourself. Testing sometimes seems to go on forever (see 'the catch-22 of testing' above), and a team that feels there's no end in sight is likely to end up despondent.

© Computer Task Group, Inc. 2007 All rights reserved.

Everyone involved in test automation knows that tests do not write themselves. Test automation is software development, which makes it subject to the same problems as any other kind of programming activity.

- Encourage developers to be positive about the testers' efforts. Developers can really help to maintain a tester's enthusiasm. This is also in their best interest, since it's likely to result in the detection of more defects before a product is released.
- Look for positives in the software. The test process encourages a negative mindset. But instead of focusing on the negatives, why not emphasize the positives? Bring some good news, for a change.
- Try to deliver bad news in a positive way. If you are given the chance, why not slip in some humor? Not long ago, we even got developers on the team enthusiastic about submitted bugs. How? We closed each bug description with a funny quote or tag line. Eventually the developers returned the favor by adding their own aphorisms and haikus to their resolution notes.
- Try to find bugs early. It's much less of an impact and much more appreciated if you find a serious bug three months before, rather than one day before a product's scheduled release.
- Temper your enthusiasm. It's okay to be enthusiastic about finding a major flaw in the system, but try to be diplomatic too.

Test Automation Paradoxes

In 2003, Brett Pettichord identified paradoxes that lie implied in the nature of test automation.[20] They include:

Bugs in automated tests

Everyone involved in test automation knows that tests do not write themselves. Test automation is software development, which makes it subject to the same problems as any other kind of programming activity. Test software should give us confidence in the production software, but what happens when our automated tests contain bugs themselves? It all comes down to the testing truism described earlier: "Who watches the watchmen?" or in this case "Who tests the automated tests?" Is our confidence in automated tests justified? Only if we have tests for our tests, which in turn should have tests… and so on. But where do we draw the line on testing tests?

Two major types of test suite failures can occur: false positives (false alarms) and false negatives (silent horrors). False alarms occur when a test is reported as failing because of errors in the test, test libraries, or test environment configuration. Silent horrors occur when an error is detected but not reported, or when a test is run incorrectly. Here are a few possible ways to mitigate the effects of the first automation paradox and avoid 'silent horrors' and 'false alarms':

- Make your test suites easy to review (clean syntax, common language).
- Verify that your automated tests also find already-known bugs.
- Use a different status for tests that can't start ('not run').
- Fix false alarms when they occur (don't let silent horrors slip in).
- If you don't have time, log the problem.

© Computer Task Group, Inc. 2007 All rights reserved.

Automated regression tests

When software changes frequently, there is a need for frequent retesting; new functionalities may introduce bugs in parts of the application that were working before. Therefore, regression tests are best automated. But the tricky part is, these software changes can also break the automated regression tests.

This is the second test automation paradox: the more the software changes, the more important regression tests become, but the greater the chance that our regression tests will be broken. In other words: our regression tests are least reliable when we need them most.

The major cause of most software project failures and of poor software quality in general is the lack of early-stage unit testing.

Developer Testing Paradox

The major cause of most software project failures and of poor software quality in general is the lack of early-stage unit testing. Almost every project post-mortem meeting—even those for successful projects—concludes that testing earlier in the process would have made a huge difference. Unit testing is, by far, the best option for improving software quality. The later a bug is found, the higher the cost of fixing it, so it is only sound economics to identify and fix bugs as early as possible. Unit testing is an opportunity to catch bugs early, before the cost of correction escalates too far.[21] (see Figure 3 below).

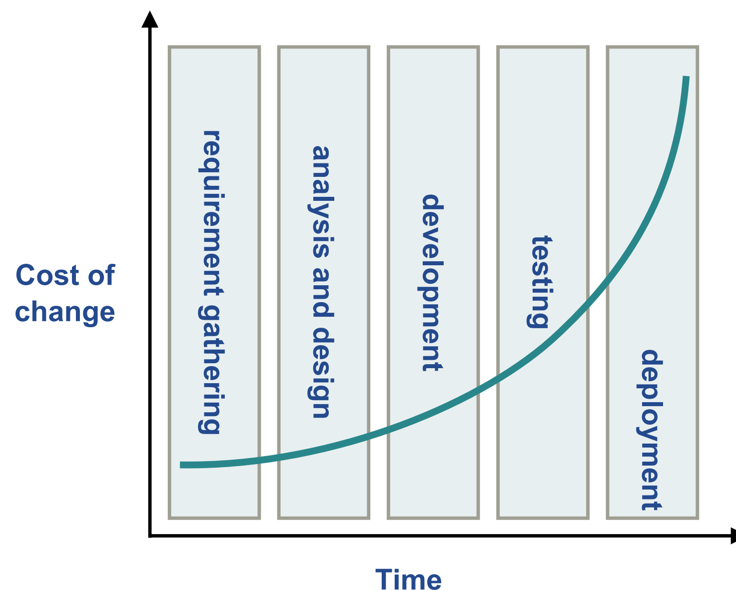

Figure 3: Boehm's cost of change curve

The benefits of unit testing are clear, and the alarming number of software project disasters and near-disasters should be enough motivation for investing in developer testing and implementing a body of unit tests in parallel with the code. Today, however, only a very small percentage of software organizations actually have a consistent developer testing policy.

© Computer Task Group, Inc. 2007 All rights reserved.

How is it possible that the practice of developer testing, which is so obviously right and so widely acknowledged as beneficial, and which could improve software quality and economics more than any other alternative, is still a rarity in software development organizations?

This discrepancy is the basis of the developer testing paradox, as defined by Alberto Savoia (2005).[22] How is it possible that the practice of developer testing, which is so obviously right and so widely acknowledged as beneficial, and which could improve software quality and economics more than any other alternative, is still a rarity in software development organizations?

One possible reason is because many heartfelt unit-testing efforts start with great enthusiasm, but end up stalling and are eventually abandoned. This is why regular, ongoing developer testing practices are the exception rather than the rule, and why there is a developer testing paradox in the first place.

Another explanation for the paradox is that several common misconceptions surround unit testing:[23]

- 'It is too time-consuming.' (You can either spend short minutes writing unit tests, or spend long hours debugging. Properly planned unit testing is actually a much more efficient use of time.)
- 'It only proves that the code does what the code does.' (The code should be tested against its specification, not against itself.)
- 'I don't need unit tests; my software will work straight away.' (Not only are testers human; developers are too. Everyone makes mistakes.)
- 'Integration tests will catch the bugs anyway.' (The earlier bugs are found, the better. Furthermore, testing won't be as thorough as it should be.)

Usability Paradox/Catch-22

The purpose of a usability test is to verify whether your software works as intended, ensuring users have a satisfying experience. It is a means for measuring how well people can use something (a computer program; a website) for its intended purpose. Usability testing is one of the most important parts of a system's development life cycle, but it is often the most overlooked because of time/budgetary issues. Investing in usability testing can be expensive, and should be thoughtfully planned to make optimal use of your always-scarce resources. But what's the optimal timing of your usability effort?

You want to plan your usability sessions early enough so that they can tell you if you're heading down the wrong direction before you're too far along. This way, changes can be made when they're least expensive. It's much easier to change a whiteboard drawing than a wireframe diagram; easier to change a wireframe diagram than a prototype; and easier to change a prototype than a finished product.

However, if usability sessions take place too early, the information you get may be made partially irrelevant because the product changes direction or conception for whatever reason. As the 'cone of uncertainty' in figure 4 on the next page shows, we know less at the beginning of a project than we do at the end. At this early stage, users can't yet interact with a finished product, so the comments and feedback received from them are potentially less relevant.

© Computer Task Group, Inc. 2007 All rights reserved.

This is the usability paradox: You need to conduct usability testing in an early stage, but at that point, there's still no working software. At a later stage, when you do have the software, you don't have the time or the money to make changes if anything serious emerges from the testing. Either way you lose. Catch-22. Or is there a way out?

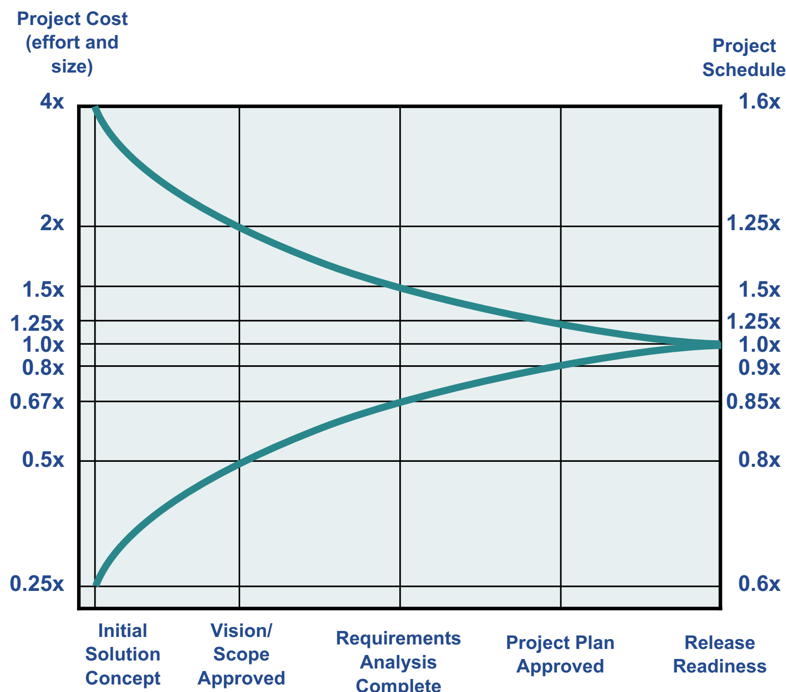

Figure 4. The cone of uncertainty (source: Microsoft.com)

Since usability involves users actually 'using' the software, it could be argued that the best time for usability testing is right at the end, with software that is almost complete. Unfortunately, at the end you can't afford to make big changes. That's exactly what happened some years ago on a test project I did. A huge amount of money was spent on design and layout studies for the software program. Testing was almost completed when we obtained the results of a usability test that was organized externally. A fair amount of users thought the application looked "horrible". That's not something you want to hear two weeks before a release!

This is the usability paradox: You need to conduct usability testing in an early stage, but at that point, there's still no working software. At a later stage, when you do have the software, you don't have the time or the money to make changes if anything serious emerges from the testing. Either way you lose. Catch-22. Or is there a way out?

The answer is probably software prototyping: that is, the process of creating an incomplete model of the future full-featured software program that can be used to give users a first idea of the completed program or allow clients to evaluate it. Also commonly used are screen generating programs that can emulate systems that don't function, but do show what the screens may look like. The main advantage of prototyping is that feedback from the users is obtained early in the project. The success of this method relies heavily on the quality of the prototype. It's also important to keep in mind that prototyping is most beneficial in systems that will have many interactions with the users. The greater the interaction between the computer and the user, the greater the benefit that can be obtained from building a quick prototype system and letting the user play with it.

© Computer Task Group, Inc. 2007 All rights reserved.

Conclusion

All the situations described in the previous chapters are somewhat contradictory and puzzling. Our ever-critical attitude is indeed a possible threat (test team paradox), and bugs in automated tests do occur (test automation paradox). The pesticide paradox, the catch-22 of testing, and the 'realities' are real. Simpson's paradox can distort our testing reports. But there's no need to let all this overwhelm us.

> *How wonderful that we have met with a paradox. Now we have some hope of making progress.*
>
> —NIELS BOHR

Paradoxes force us to question our basic assumptions and find a different context in which the contradictory ideas make sense. And in the process, we do some thinking. Indeed, the very act of 'seeing' the paradox—the ability to entertain two contradictory ideas at the same time—lies at the heart of creative thinking.

This quote of the Danish Nobel prize-winner Niels Bohr captures the essence of paradoxes: they force us to question our basic assumptions and find a different context in which the contradictory ideas make sense. And in the process, we do some thinking. Indeed, the very act of 'seeing' the paradox—the ability to entertain two contradictory ideas at the same time—lies at the heart of creative thinking.

The purpose of paradoxes is to arrest attention and to provoke fresh thought. In science, this process frequently leads to major breakthroughs. While I don't claim that the testing paradoxes and catch-22s described above will lead to major breakthroughs within testing, I would argue that becoming aware of them leads us to think about the matter—and thinking may lead to creative rethinking. Maybe we can expand our focus to look at options or possibilities that we normally wouldn't consider. Maybe that process will turn a potentially negative outcome into something positive.

Or—maybe we should just view these phenomena as little particles of knowledge that can help us put some aspects of the overall software testing process into clearer perspective. As Sir Francis Bacon said centuries ago: "Knowledge is power." It gives us the power to beat the odds, to get a better understanding of our profession and—hopefully—to become better at what we do.

© Computer Task Group, Inc. 2007 All rights reserved.

References

- Wikipedia on paradoxes *(http://www.wikipedia.org)*
- Raymond Smullyan, "What is the Name of This Book?", 1978
- Lee Thomas, "Quality versus speed: Paradox lost?", *IBM developerworks,* 2003
- Ingo Melzer, *Testing of a Computer Program on the Example of a Medical Application with diversification and other Methods,* 1996
- Yang, M.C.K.; Chao, A. "Reliability-estimation and stopping-rules for software testing, based on repeated appearances of bugs"; *IEEE Transactions on Reliability*, vol.44, no.2, p. 315-21, 1995
- Boris Beizer, *Software Testing Techniques,* Second Edition, 1990
- Ron Patton, *Testing Axioms,* 2000
- Nihit Kaul, weblog, 2004
- James Bach, "Test Automation Snake Oil", *Windows Tech Journal,* October 1996
- Michael Hunter, "Five questions with Sara Ford", *Thoughts from a Braidy Tester,* 2007
- Hans Schaefer, "What a tester should know, even after midnight", 2006
- Jiantao Pan, *Software Testing,* 1999 *(http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/)*
- Scientist Live, Failure breeds success, 2007 (University of Exeter)
- Darren Roberts, "Ten excellent ways how failure can be more beneficial to you than success", 1998
- Matthew Heusser, "Classic mistakes in testing: revisited", 2005
- James Bach's blog, "How to investigate intermittent problems", 2005
- Joel Spolsky, "What is the work of dogs in this country?", *JoelOnSoftware.com,* 2001
- Mike Kelly, "Four classic problems with scripted testing", 2006
- James McCaffrey, "Software Testing paradoxes", December 2005
- D. Braess, "On a paradox of traffic planning", *The Journal of Transportation Science, volume 39, 2005,* pp. 446-450
- "The virtual center for supernetworks", *(http://supernet.som.umass.edu/facts/braess.html)*
- William Echlin, "The Test Team Paradox", *www.stickyminds.com,* 2006
- Alberto Savoia, "The Developer Testing Paradox", 2005
- B.W. Boehm, "Software Engineering Economics", 1980
- IPL, "Why bother to unit test?", 1997
- B. Pettichord, "Three paradoxes of Test Automation", 2003
- J. Kohl, "Test-Driven Development from a Conventional Software Testing Perspective, Part 2", 2006
- L. Becker, "90% of all usablity testing is useless", 2004

© Computer Task Group, Inc. 2007 All rights reserved.

Backed by over 40 years of experience, CTG provides IT staffing, application management outsourcing, consulting, and software development and integration solutions to help companies focus on their core businesses and use IT as a competitive advantage to excel in their markets. CTG combines in-depth understanding of our clients' businesses with a full range of integrated services and proprietary ISO 9001:2000-certified service methodologies. Our IT professionals, based in an international network of offices in North America and Europe, have a proven track record of delivering solutions that work.

More information about CTG is available on the Web at www.ctg.com.

For more information about CTG's Testing Services, please contact:

In Europe:

CTG Belgium NV/SA
Global Testing Practice
Woluwelaan 140A bus 3,
1831 Diegem, Belgium
32 (0)2 720 51 70
Fax: + 32 (0)2 725 09 20
info.be@ctg.com.

In the U.S.:

CTG Inc.
Global Testing Practice
800 Delaware Ave.
Buffalo, NY 14209
716/882-8000
Fax: 716/887-7456
info.us@ctg.com

- B. Marick, Exploration through example, Test planning documents *(http://www.exampler.com/blog/2007/07/19/test-planning-documents/)*
- John Crinnion: *Evolutionary Systems Development, a practical guide to the use of prototyping within a structured systems methodology.* Plenum Press, New York, 1991. Page 18.
- Wikipedia on prototyping (http://www.wikipedia.org)

Notes

1. Lee Thomas, "Quality versus speed: Paradox lost?", IBM developerworks, 2003
2. Source:Paradox, Wikipedia
3. Raymond Smullyan, "What's the name of this book", 1978
4. Ingo Melzer, "Testing of a Computer Program on the Example of a Medical Application with Diversification and Other Methods" 1996
5. Yang, M.C.K.; Chao, A. "Reliability-estimation and stopping-rules for software testing, based on repeated appearances of bugs"; *IEEE Transactions on Reliability,* vol.44, no.2, p. 315-21, 1995
6. Michael Hunter, Five questions with Sara Ford", *Thoughts from a Braidy Tester,* 2007
7. Hans Schaefer, "What a tester should know, even after midnight", 2006
8. Mike Kelly, "Four Classic Problems with Scripted Testing", 2006
9. Joel Spolsky, "What is the work of dogs in this country?", *JoelOnSoftware.com,* 2001
10. Matthew Heusser, "Classic mistakes in testing: revisited", 2005
11. Ron Patton, *Testing Axioms,* 2000
12. *The Complexity Barrier,* (Beizer, 1990): Software complexity (and therefore that of bugs) grows to the limit of our ability to manage that complexity
13. Scientist Live, Failure breeds success, 2007 (University of Exeter)
14. James Bach's blog, "How to investigate intermittent problems", 2005
15. Hans Schaefer, "What a tester should know, even after midnight", 2006
16. Boris Beizer, *Software Testing Techniques,* Second Edition, 1990
17. James Bach, "Test Automation Snake Oil", *Windows Tech Journal,* October 1996
18. Nihit Kaul, weblog, 2004
19. James McCaffrey, "Software Testing paradoxes", December 2005
20. B. Pettichord, "Three paradoxes of test automation", 2003
21. B. W. Boehm, "Software Engineering Economics", 1980
22. Alberto Savoia, "The Developer Testing Paradox", 2005
23. IPL, "Why bother to unit test?", 1997

© Computer Task Group, Inc. 2007 All rights reserved.