Classic Testing Mistakes

Brian Marick
Testing Foundations
marick@testing.com

It's easy to make mistakes when testing software or planning a testing effort. Some mistakes are made so often, so repeatedly, by so many different people, that they deserve the label Classic Mistake.

Classic mistakes cluster usefully into five groups, which I've called "themes":

- The Role of Testing: who does the testing team serve, and how does it do that?
- Planning the Testing Effort: how should the whole team's work be organized?
- Personnel Issues: who should test?
- The Tester at Work: designing, writing, and maintaining individual tests.
- Technology Rampant: quick technological fixes for hard problems.

I have two goals for this paper. First, it should identify the mistakes, put them in context, describe why they're mistakes, and suggest alternatives. Because the context of one mistake is usually prior mistakes, the paper is written in a narrative style rather than as a list that can be read in any order. Second, the paper should be a handy checklist of mistakes. For that reason, the classic mistakes are printed in a larger bold font when they appear in the text, and they're also summarized at the end.

Although many of these mistakes apply to all types of software projects, my specific focus is the testing of commercial software products, not custom software or software that is safety critical or mission critical.

This paper is essentially a series of bug reports for the testing process. You may think some of them are features, not bugs. You may disagree with the severities I assign. You may want more information to help in debugging, or want to volunteer information of your own. Any decent bug reporting system will treat the original bug report as the first part of a conversation. So should it be with this paper. Therefore, see http://www.stlabs.com/marick/classic.htm for an ongoing discussion of this topic.

Theme One: The Role of Testing

A first major mistake people make is thinking that **the testing team is responsible for assuring quality.** This role, often assigned to the first testing team in an organization, makes it the last defense, the barrier between the development team (accused of producing bad quality) and the customer (who must be protected from them). It's characterized by a testing team (often called the "Quality Assurance Group") that has

formal authority to prevent shipment of the product. That in itself is a disheartening task: the testing team can't improve quality, only enforce a minimal level. Worse, that authority is usually more apparent than real. Discovering that, together with the perverse incentives of telling developers that quality is someone else's job, leads to testing teams and testers who are disillusioned, cynical, and view themselves as victims. We've learned from Deming and others that products are better and cheaper to produce when everyone, at every stage in development, is responsible for the quality of their work ([Deming86], [Ishikawa85]).

In practice, whatever the formal role, most organizations believe that **the purpose of testing is to find bugs**. This is a less pernicious definition than the previous one, but it's missing a key word. When I talk to programmers and development managers about testers, one key sentence keeps coming up: "**Testers aren't finding the <u>important</u> bugs.**" Sometimes that's just griping, sometimes it's because the programmers have a skewed sense of what's important, but I regret to say that all too often it's valid criticism. Too many bug reports from testers are minor or irrelevant, and too many important bugs are missed.

What's an important bug? Important to whom? To a first approximation, the answer must be "to customers". Almost everyone will nod their head upon hearing this definition, but do they mean it? Here's a test of your organization's maturity. Suppose your product is a system that accepts email requests for service. As soon as a request is received, it sends a reply that says "your request of 5/12/97 was accepted and its reference ID is NIC-051297-3". A tester who sends in many requests per day finds she has difficulty keeping track of which request goes with which ID. She wishes that the original request were appended to the acknowledgement. Furthermore, she realizes that some customers will also generate many requests per day, so would also appreciate this feature. Would she:

1. file a bug report documenting a usability problem, with the expectation that it will be assigned a reasonably high priority (because the fix is clearly useful to everyone, important to some users, and easy to do)?

2. file a bug report with the expectation that it will be assigned "enhancement request" priority and disappear forever into the bug database?

3. file a bug report that yields a "works as designed" resolution code, perhaps with an email "nastygram" from a programmer or the development manager?

4. not bother with a bug report because it would end up in cases (2) or (3)?

If **usability problems are not considered valid bugs,** your project defines the testing task too narrowly. Testers are restricted to checking whether the product does what was intended, not whether what was intended is useful. Customers do not care about the distinction, and testers shouldn't either.

Testers are often the only people in the organization who use the system as heavily as an expert. They notice usability problems that experts will see. (Formal usability testing almost invariably concentrates on novice users.) Expert customers often don't report

usability problems, because they've been trained to know it's not worth their time. Instead, they wait (in vain, perhaps) for a more usable product and switch to it. Testers can prevent that lost revenue.

While defining the purpose of testing as "finding bugs important to customers" is a step forward, it's more restrictive than I like. It means that there is **no focus on an estimate of quality (and on the quality of that estimate)**. Consider these two situations for a product with five subsystems.

1. 100 bugs are found in subsystem 1 before release. (For simplicity, assume that all bugs are of the highest priority.) No bugs are found in the other subsystems. After release, no bugs are reported in subsystem 1, but 12 bugs are found in each of the other subsystems.
2. Before release, 50 bugs are found in subsystem 1. 6 bugs are found in each of the other subsystems. After release, 50 bugs are found in subsystem 1 and 6 bugs in each of the other subsystems.

From the "find important bugs" standpoint, the first testing effort was superior. It found 100 bugs before release, whereas the second found only 74. But I think you can make a strong case that the second effort is more useful in practical terms. Let me restate the two situations in terms of what a test manager might say before release:

1. "We have tested subsystem 1 very thoroughly, and we believe we've found almost all of the priority 1 bugs. Unfortunately, we don't know anything about the bugginess of the remaining five subsystems."
2. "We've tested all subsystems moderately thoroughly. Subsystem 1 is still very buggy. The other subsystems are about 1/10th as buggy, though we're sure bugs remain."

This is, admittedly, an extreme example, but it demonstrates an important point. The project manager has a tough decision: would it be better to hold on to the product for more work, or should it be shipped now? Many factors - all rough estimates of possible futures - have to be weighed: Will a competitor beat us to release and tie up the market? Will dropping an unfinished feature to make it into a particular magazine's special "Java Development Environments" issue cause us to suffer in the review? Will critical customer X be more annoyed by a schedule slip or by a shaky product? Will the product be buggy enough that profits will be eaten up by support costs or, worse, a recall? [1]

The testing team will serve the project manager better if it concentrates first on providing estimates of product bugginess (reducing uncertainty), then on finding more of the bugs that are estimated to be there. That affects test planning, the topic of the next theme.

It also affects status reporting. Test managers often err by **reporting bug data without putting it into context.** Without context, project management tends to focus on one graph:

---

[1] Notice how none of the decisions depend solely on the product's bugginess. That's another reason why giving the testing manager "stop ship" authority is a bad idea. He or she simply doesn't have enough information to use that authority wisely. The project manager might not have enough either, but won't have <u>less</u>.

Bug Status
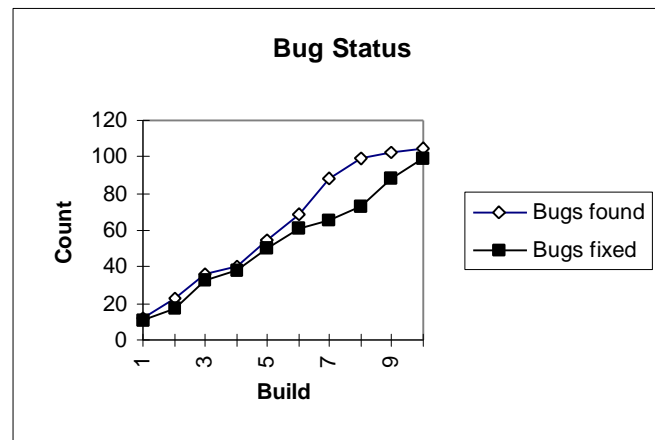
The flattening in the curve of bugs found will be interpreted in the most optimistic possible way unless you as test manager explain the limitations of the data:

- "Only half the planned testing tasks have been finished, so little is known about half the areas in the project. There could soon be a big spike in the number of bugs found."
- "That's especially likely because the last two weekly builds have been lightly tested. I told the testers to take their vacations now, before the project hits crunch mode."
- "Furthermore, based on previous projects with similar amounts and kinds of testing effort, it's reasonable to expect at least 45 priority-1 bugs remain undiscovered. Historically, that's pretty high for a successful product."

For discussions of using bug data, see [Cusumano95], [Rothman96], and [Marick97].

Earlier I asserted that testers can't directly improve quality; they can only measure it. That's true only if you find yourself **starting testing too late**. Tests designed before coding begins can improve quality. They inform the developer of the kinds of tests that will be run, including the special cases that will be checked. The developer can use that information while thinking about the design, during design inspections, and in his own developer testing.[2]

Early test design can do more than prevent coding bugs. As will be discussed in the next theme, many tests will represent user tasks. The process of designing them can find user interface and usability problems before expensive rework is required. I've found problems like no user-visible place for error messages to go, pluggable modules that didn't fit

---

[2] One person who worked in a pathologically broken organization told me that they were given the acceptance test in advance. They coded the program to recognize the test cases and return the correct answer, bypassing completely the logic that was supposed to calculate the answer. Few companies are that bad, but you could argue that programmers will tend to produce code "trained" for the tests. If the tests are good, that's not a problem - the code is also trained for the real customers. The biggest danger is that the programmers will interpret the tests as narrow special cases, rather than handling the more general situation. That can be forestalled by writing the early test designs in terms of general situations rather than specific inputs: "more than two columns per page" rather than "three two-inch columns on an A4 page". Also, the tests given to the programmers will likely be supplemented by others designed later.

together, two screens that had to be used together but could not be displayed simultaneously, and "obvious" functions that couldn't be performed. Test design fits nicely into any usability engineering effort ([Nielsen93]) as a way of finding specification bugs.

I should note that involving testing early feels unnatural to many programmers and development managers. There may be feelings that you are intruding on their turf or not giving them the chance to make the mistakes that are an essential part of design. Take care, especially at first, not to increase their workload or slow them down. It may take one or two entire projects to establish your credibility and usefulness.

Theme Two: Planning the Testing Effort

I'll first discuss specific planning mistakes, then relate test planning to the role of testing.

It's not unusual to see **test plans biased toward functional testing.** In functional testing, particular features are tested in isolation. In a word processor, all the options for printing would be applied, one after the other. Editing options would later get their own set of tests.

But there are often interactions between features, and functional testing tends to miss them. For example, you might never notice that the sequence of operations `open a document, edit the document, print the whole document, edit one page, print that page` doesn't work. But customers surely will, because they don't use products functionally. They have a task orientation. To find the bugs that customers see - that are important to customers - you need to write tests that cross functional areas by mimicking typical user tasks. This type of testing is called scenario testing, task-based testing, or use-case testing.

A bias toward functional testing also **underemphasizes configuration testing.** Configuration testing checks how the product works on different hardware and when combined with different third party software. There are typically many combinations that need to be tried, requiring expensive labs stocked with hardware and much time spent setting up tests, so configuration testing isn't cheap. But, it's worth it when you discover that your standard in-house platform which "entirely conforms to industry standards" actually behaves differently from most of the machines on the market.

Both configuration testing and scenario testing test global, cross-functional aspects of the product. Another type of testing that spans the product checks how it behaves under stress (a large number of transactions, very large transactions, a large number of simultaneous transactions). **Putting stress and load testing off to the last minute** is common, but it leaves you little time to do anything substantive when you discover your product doesn't scale up to more than 12 users.[3]

---

[3] Failure to apply particular types of testing is another reason why developers complain that testers aren't finding the important bugs. Developers of an operating system could be spending all their time debugging crashes of their private machines, crashes due to networking bugs under normal load. The testers are doing straight "functional

Two related mistakes are **not testing the documentation** and **not testing installation procedures.** Testing the documentation means checking that all the procedures and examples in the documentation work. Testing installation procedures is a good way to avoid making a bad first impression.

How about avoiding testing altogether?

At a conference last year, I met (separately) two depressed testers who told me their management was of the opinion that the World Wide Web could reduce testing costs. "Look at [wildly successful internet company]. They distribute betas over the network and get their *customers* to do the testing for free!" The Windows 95 beta program is also cited in similar ways.

Beware of **an overreliance on beta testing**. Beta testing seems to give you test cases representative of customer use - because the test cases are customer use. Also, bugs reported by customers are by definition those important to customers. However, there are several problems:

1. The customers probably aren't that representative. In the common high-tech marketing model[4], beta users, especially those of the "put it on your web site and they will download" sort, are the early adopters, those who like to tinker with new technologies. They are not the pragmatists, those who want to wait until the technology is proven and safe to adopt. The usage patterns of these two groups are different, *as are the kinds of bugs they consider important*. In particular, early adopters have a high tolerance for bugs with workarounds and for bugs that "just go away" when they reload the program. Pragmatists, who are much less tolerant, make up the large majority of the market.

2. Even of those beta users who actually use the product, most will not use it seriously. They will give it the equivalent of a quick test drive, rather than taking the whole family for a two week vacation. As any car buyer knows, the test drive often leaves unpleasant features undiscovered.

3. Beta users - just like customers in general - don't report usability problems unless prompted. They simply silently decide they won't buy the final version.

4. Beta users - just like customers in general - often won't report a bug, especially if they're not sure what they did to cause it, or if they think it is obvious enough that someone else must have already reported it.

5. When beta users report a bug, the bug report is often unusable. It costs much more time and effort to handle a user bug report than one generated internally.

---

tests" on isolated machines, so they don't find bugs. The bugs they do find are not more serious than crashes (usually defined as highest severity for operating systems), and they're probably less.

[4] See [Moore91] or [Moore95]. I briefly describe this model in a review of Moore's books, available through Pure Atria's book review pages (http://www.pureatria.com).

Beta programs can be useful, but they require careful planning and monitoring if they are to do more than give a warm fuzzy feeling that at least some customers have used the product before it's inflicted on all of them. See [Kaner93] for a brief description.

The one situation in which beta programs are unequivocally useful is in configuration testing. For any possible screwy configuration, you can find a beta user who has it. You can do much more configuration testing than would be possible in an in-house lab (or even perhaps an outsourced testing agency). Beta users won't do as thorough a job as a trained tester, but they'll catch gross errors of the "BackupBuster doesn't work on this brand of 'compatible' floppy tape drive" sort.

Beta programs are also useful for building word of mouth advertising, getting "first glance" reviews in magazines, supporting third-party vendors who will build their product on top of yours, and so on. Those are properly marketing activities, not testing.

Planning and replanning in support of the role of testing

Each of the types of testing described above, including functional testing, reduces uncertainty about a particular aspect of the product. When done, you have confidence that some functional areas are less buggy, others more. The product either usually works on new configurations, or it doesn't.[5]

There's a natural tendency toward **finishing one testing task before moving on to the next,** but that may lead you to discover bad news too late. It's better to know something about all areas than everything about a few. When you've discovered where the problem areas lie, you can test them to greater depth as a way of helping the developers raise the quality by finding the important bugs.[6]

Strictly, I've been over-simplistic in describing testing's role as reducing uncertainty. It would be better to say "risk-weighted uncertainty". Some areas in the product are riskier than others, perhaps because they're used by more customers or because failures in that area would be particularly severe. Riskier areas require more certainty. **Failing to correctly identify risky areas** is a common mistake, and it leads to misallocated testing effort. There are two sound approaches for identifying risky areas:

1. Ask everyone you can for their opinion. Gather data from developers, marketers, technical writers, customer support people, and whatever customer representatives

---

[5] I use "confidence" in its colloquial rather than its statistical sense. Conventional testing that searches specifically for bugs does not allow you to make statements like "this product will run on 95±5% of Wintel machines". In that sense, it's weaker than statistical or reliability testing, which uses statistical profiles of the customer environment to both find bugs and make failure estimates. (See [Dyer92], [Lyu96], and [Musa87].) Statistical testing can be difficult to apply, so I concentrate on a search for bugs as the way to get a usable estimate. A lack of statistical validity doesn't mean that bug numbers give you nothing but "warm and fuzzy (or cold and clammy) feelings". Given a modestly stable testing process, development process, and product line, bug numbers lead to distinctly better decisions, even if they don't come with p-values or statistical confidence intervals.

[6] It's expensive to test quality into the product, but it may be the only alternative. Code redesigns and rewrites may not be an option.

you can find. See [Kaner96a] for a good description of this kind of collaborative test planning.

2. Use historical data. Analyzing bug reports from past products (especially those from customers, but also internal bug reports) helps tell you what areas to explore in this project.

"So, winter's early this year. We're <u>still</u> going to invade Russia."

Good testers are systematic and organized, yet they are exposed to all the chaos and twists and turns and changes of plan typical of a software development project. In fact, the chaos is magnified by the time it gets to testers, because of their position at the end of the food chain and typically low status.[7]   One unfortunate reaction is **sticking stubbornly to the test plan**. Emotionally, this can be very satisfying: "They can flail around however they like, but I'm going to hunker down and do my job." The problem is that your job is not to write tests. It's to find the bugs that matter in the areas of greatest uncertainty and risk, and ignoring changes in the reality of the product and project can mean that your testing becomes irrelevant.[8]

That's not to say that testers should jump to readjust all their plans whenever there's a shift in the wind, but my experience is that more testers let their plans fossilize than overreact to project change.

Theme Three: Personnel Issues

Fresh out of college, I got my first job as a tester. I had been hired as a developer, and knew nothing about testing, but, as they said, "we don't know enough about you yet, so we'll put you somewhere where you can't do too much damage". In due course, I "graduated" to development.

Using testing as a transitional job for new programmers is one of the two classic mistaken ways to staff a testing organization. It has some virtues. One is that you really <u>can</u> keep bad hires away from the code. A bozo in testing is often less dangerous than a bozo in development. Another is that the developer <u>may</u> learn something about testing that will be useful later. (In my case, it founded a career.) And it's a way for the new hire to learn the product while still doing some useful work.

The advantages are outweighed by the disadvantage: the new hire can't wait to get out of testing. That's hardly conducive to good work. You could argue that the testers have to do good work to get "paroled". Unfortunately, because people tend to be as impressed by effort as by results, vigorous activity - especially activity that establishes credentials as a

---

[7] How many proposed changes to a product are rejected because of their effect on the testing schedule? How often does the effect on the testing team even cross a developer's or marketer's mind?

[8] This is yet another reason why developers complain that testers aren't finding the important bugs. Because of market pressure, the project has shifted to an Internet focus, but the testers are still using and testing the old "legacy" interface instead of the now critically important web browser interface.

programmer - becomes the way out. As a result, the fledgling tester does things like become the expert in the local programmable editor or complicated freeware tool. That, at least, is a potentially useful role, though it has nothing to do with testing. More dangerous is vigorous but misdirected testing activity; namely, test automation. (See the last theme.)

Even if novice testers were well guided, having so much of the testing staff be transients could only work if testing is a shallow algorithmic discipline. In fact, good testers require deep knowledge and experience.

The second classic mistake is **recruiting testers from the ranks of failed programmers**. There are plenty of good testers who are not good programmers, but a bad programmer likely has some work habits that will make him a bad tester, too. For example, someone who <u>makes</u> lots of bugs because he's inattentive to detail will <u>miss</u> lots of bugs for the same reason.

So how should the testing team be staffed? If you're willing to be part of the training department, go ahead and accept new programmer hires.[9] Accept as applicants programmers who you suspect are rejects (some fraction of them really have gotten tired of programming and want a change) but interview them as you would an outside hire. When interviewing, concentrate less on formal qualifications than on intelligence and the character of the candidate's thought. A good tester has these qualities:[10]

- methodical and systematic.
- tactful and diplomatic (but firm when necessary).
- skeptical, especially about assumptions, and wants to see concrete evidence.
- able to notice and pursue odd details.
- good written and verbal skills (for explaining bugs clearly and concisely).
- a knack for anticipating what others are likely to misunderstand. (This is useful both in finding bugs and writing bug reports.)
- a willingness to get one's hands dirty, to experiment, to try something to see what happens.

Be especially careful to avoid the trap of **testers who are not domain experts.** Too often, the tester of an accounting package knows little about accounting. Consequently, she finds bugs that are unimportant to accountants and misses ones that are. Further, she writes bug reports that make serious bugs seem irrelevant. A programmer may not see past the unrepresentative test to the underlying important problem. (See the discussion of reporting bugs in the next theme.)

Domain experts may be hard to find. Try to find a few. And hire testers who are quick studies and are good at understanding other people's work patterns.

---

[9] Some organizations rotate all developers through testing. Well, all developers except those with enough clout to refuse. And sometimes people not in great demand don't seem ever to rotate out. I've seen this approach work, but it's fragile.

[10] See also the list in [Kaner93], chapter 15.

Two groups of people are readily at hand and often have those skills. But testing teams often **do not seek out applicants from the customer service staff or the technical writing staff.** The people who field email or phone problem reports develop, if they're good, a sense of what matters to the customer (at least to the vocal customer) and the best are very quick on their mental feet.

Like testers, technical writers often also lack detailed domain knowledge. However, they're in the business of translating a product's behavior into terms that make sense to a user. Good technical writers develop a sense of what's important, what's confusing, and so on. Those areas that are hard to explain are often fruitful sources of bugs. (What confuses the user often also confuses the programmer.)

One reason these two groups are not tapped is **an insistence that testers be able to program.** Programming skill brings with it certain advantages in bug hunting. A programmer is more likely to find the number 2,147,483,648 interesting than an accountant will. (It overflows a signed integer on most machines.) But such tricks of the trade are easily learned by competent non-programmers, so not having them is a weak reason for turning someone down.

If you hire according to these guidelines, you will avoid **a testing team that lacks diversity.** All of the members will lack some skills, but the team as a whole will have them all. Over time, in a team with mutual respect, the non-programmers will pick up essential tidbits of programming knowledge, the programmers will pick up domain knowledge, and the people with a writing background will teach the others how to deconstruct documents.

All testers - but non-programmers especially - will be hampered by a **physical separation between developers and testers.** A smooth working relationship between developers and testers is essential to efficient testing. Too much valuable information is unwritten; the tester finds it by talking to developers. Developers and testers must often work together in debugging; that's much harder to do remotely. Developers often dismiss bug reports too readily, but it's harder to do that to a tester you eat lunch with.

Remote testing can be made to work - I've done it - but you have to be careful. Budget money for frequent working visits, and pay attention to interpersonal issues.

Some believe that **programmers can't test their own code**. On the face of it, this is false: programmers test their code all the time, and they do find bugs. Just not enough of them, which is why we need independent testers.

But if independent testers are testing, and programmers are testing (and inspecting), isn't there a potential duplication of effort? And isn't that wasteful? I think the answer is yes. Ideally, programmers would concentrate on the types of bugs they can find adequately well, and independent testers would concentrate on the rest.

The bugs programmers can find well are those where their code does not do what they intended. For example, a reasonably trained, reasonably motivated programmer can do a perfectly fine job finding boundary conditions and checking whether each known equivalence class is handled. What programmers do poorly is discovering overlooked special cases (especially error cases), bugs due to the interaction of their code with other people's code (including system-wide properties like deadlocks and performance problems), and usability problems.

Crudely put, good programmers do functional testing, and testers should do everything else.[11] Recall that I earlier claimed an over-concentration on functional testing is a classic mistake. Decent programmer testing magnifies the damage it does.

Of course, decent programmer testing is relatively rare, because **programmers are neither trained nor motivated to test.** This is changing, gradually, as companies realize it's cheaper to have bugs found and fixed quickly by one person, instead of more slowly by two. Until then, testers must do both the testing that programmers can do and the testing only testers can do, but must take care not to let functional testing squeeze out the rest.

Theme Four: The Tester At Work

When testing, you must decide how to exercise the program, then do it. The doing is ever so much more interesting than the deciding. A tester's itch to start breaking the program is as strong as a programmer's itch to start writing code - and it has the same effect: design work is skimped, and quality suffers. **Paying more attention to running tests than to designing them** is a classic mistake. A tester who is not systematic, who does not spend time laying out the possibilities in advance, will overlook special cases. They may be the same subtle ones that the programmers overlooked.

Concentration on execution also results in **unreviewed test designs**. Just like programmers, testers can benefit from a second pair of eyes. Reviews of test designs needn't be as elaborate as product design reviews, but a short check of the testing approach and the resulting tests can find significant omissions at low cost.

What is a test design?

A test design should contain a description of the setup (including machine configuration for a configuration test), inputs given to the product, and a description of expected results. One common mistake is **being too specific about test inputs and procedures**.

Let's assume manual test implementation for the moment. A related argument for automated tests will be discussed in the next section. Suppose you're testing a banking application. Here are two possible test designs:

---

[11] Independent testers will also provide a "safety net" for programmer testing. A certain amount of functional testing might be planned, or it might be a side effect of the other types of testing being done.

Design 1

Setup: initialize the balance in account 12 with $100.
Procedure:
  Start the program.
  Type 12 in the Account window.
  Press OK.
  Click on the 'Withdraw' toolbar button.
  In the withdraw popup dialog, click on the 'all' button.
  Press OK.
  Expect to see a confirmation popup that says "You are about to withdraw all the money from this account. Continue?"
  Press OK.
  Expect to see a 0 balance in the account window.
  Separately query the database to check that the zero balance has been posted.
  Exit the program with File->Exit.

Design 2

Setup: initialize the balance with a positive value.
Procedure:
  Start the program on that account.
  Withdraw all the money from the account using the 'all' button.
  It's an error if the transaction happens without a confirmation popup.
  Immediately thereafter:
  - Expect a $0 balance to be displayed.
  - Independently query the database to check that the zero balance has been posted.

The first design style has these advantages:

- The test will always be run the same way. You are more likely to be able to reproduce the bug. So will the programmer.
- It details all the important expected results to check. Imprecise expected results make failures harder to notice. For example, a tester using the second style would find it easier to overlook a spelling error in the confirmation popup, or even that it was the wrong popup.
- Unlike the second style, you always know exactly what you've tested. In the second style, you couldn't be sure that you'd ever gotten to the Withdraw dialog via the toolbar. Maybe the menu was always used. Maybe the toolbar button doesn't work at all!
- By spelling out all inputs, the first style prevents testers from carelessly overusing simple values. For example, a tester might always test accounts with $100, rather than using a variety of small and large balances. (Either style should include explicit tests for boundary and special values.)

However, there are also some disadvantages:

- The first style is more expensive to create.

- The inevitable minor changes to the user interface will break it, so it's more expensive to maintain.
- Because each run of the test is exactly the same, there's no chance that a variation in procedure will stumble across a bug.
- It's hard for testers to follow a procedure exactly. When one makes a mistake - pushes the wrong button, for example - will she really start over?

On balance, I believe the negatives often outweigh the positives, provided there is a separate testing task to check that all the menu items and toolbar buttons are hooked up. (Not only is a separate task more efficient, it's less error-prone. You're less likely to accidentally omit some buttons.)

I do not mean to suggest that test cases should not be rigorous, only that they should be no more rigorous than is justified, and that we testers sometimes error on the side of uneconomical detail.

Detail in the expected results is less problematic than in the test procedure, but too much detail can focus the tester's attention too much on checking against the script he's following. That might encourage another classic mistake: **not noticing and exploring "irrelevant" oddities.** Good testers are masters at noticing "something funny" and acting on it. Perhaps there's a brief flicker in some toolbar button which, when investigated, reveals a crash. Perhaps an operation takes an oddly long time, which suggests to the attentive tester that increasing the size of an "irrelevant" dataset might cause the program to slow to a crawl. Good testing is a combination of following a script and using it as a jumping-off point for an exploration of the product.

An important special case of overlooking bugs is **checking that the product does what it's supposed to do, but not that it doesn't do what it isn't supposed to do**. As an example, suppose you have a program that updates a health care service's database of family records. A test adds a second child to Dawn Marick's record. Almost all testers would check that, after the update, Dawn now has two children. Some testers - those who are clever, experienced, or subject matter experts - would check that Dawn Marick's spouse, Brian Marick, also now has two children. Relatively few testers would check that no one else in the database has had a child added. They would miss a bug where the programmer over-generalized and assumed that all "family information" updates should be applied both to a patient and to all members of her family, giving Paul Marick (aged 2) a child.

Ideally, every test should check that all data that should be modified has been modified and that all other data has been unchanged. With forethought, that can be built into automated tests. Complete checking may be impractical for manual tests, but occasional quick scans for data that might be corrupted can be valuable.

Testing should not be isolated work

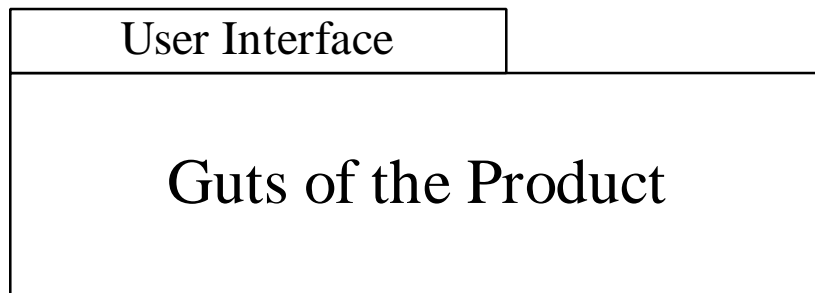Here's another version of the test we've been discussing:

Design 3

Withdraw all with confirmation and normal check for 0.

That means the same thing as Design 2 - but only to the original author. **Test suites that are understandable only by their owners** are ubiquitous. They cause many problems when their owners leave the company; sometimes many month's worth of work has to be thrown out.

I should note that designs as detailed as Designs 1 or 2 often suffer a similar problem. Although they can be run by anyone, not everyone can update them when the product's interface changes. Because the tests do not list their purposes explicitly, updates can easily make them test a little less than they used to. (Consider, for example, a suite of tests in the Design 1 style: how hard will it be to make sure that all the user interface controls are touched in the revised tests? Will the tester even know that's a goal of the suite?) Over time, this leads to what I call "test suite decay," in which a suite full of tests runs but no longer tests much of anything at all.[12]

Another classic mistake involves the boundary between the tester and programmer. Some products are mostly user interface; everything they do is visible on the screen. Other products are mostly internals; the user interface is a "thin pipe" that shows little of what happens inside. The problem is that testing has to use that thin pipe to discover failures. What if complicated internal processing produces only a "yes or no" answer? Any given test case could trigger many internal faults that, through sheer bad luck, don't produce the wrong answer.[13]

In such situations, testers sometimes rely solely on programmer ("unit") testing. In cases where that's not enough, **testing only through the user-visible interface** is a mistake. It is far better to get the programmers to add "testability hooks" or "testpoints" that reveal selected internal state. In essence, they convert a product like this:

```
┌─────────────────────────┐
│     User Interface       │
├─────────────────────────┴────────┐
│                                   │
│       Guts of the Product         │
│                                   │
│                                   │
└───────────────────────────────────┘
```

to one like this:

---

[12] The purpose doesn't need to be listed with the test. It may be better to have a central document describing the purposes of a group of tests, perhaps in tabular form. Of course, then you have to keep that document up to date.

[13] This is an example of the formal notion of "testability". See, [Friedman95] or [Voas91] for an academic treatment.

	Testing
User Interface	Interface
Guts of the Product	
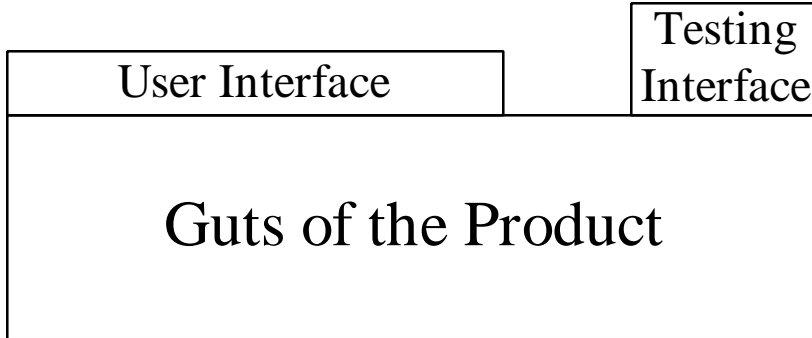
It is often difficult to convince programmers to add test support code to the product. (Actual quote: "I don't want to clutter up my code with testing crud.") Persevere, start modestly, and take advantage of these facts:

1. The test support code is often a simple extension of the debugging support code programmers write anyway.[14]

2. A small amount of test support code often goes a long way.

A common objection to this approach is that the test support code must be compiled out of the final product (to avoid slowing it down). If so, tests that use the testing interface "aren't testing what we ship". It is true that some of the tests won't run on the final version, so you may miss bugs. But, without testability code, you'll miss bugs that don't reveal themselves through the user interface. It's a risk tradeoff, and I believe that adding test support code usually wins. See [Marick95], chapter 13, for more details.

In one case, there's an alternative to having the programmer add code to the product: have a tool do it. Commercial tools like Purify, Boundschecker, and Sentinel automatically add code that checks for certain classes of failures (such as memory leaks).[15] They provide a narrow, specialized testing interface. For marketing reasons, these tools are sold as programmer debugging tools, but they're equally test support tools, and I'm amazed that testing groups don't use them as a matter of course.

Testability problems are exacerbated in distributed systems like conventional client/server systems, multi-tiered client/server systems, Java applets that provide smart front-ends to web sites, and so forth. Too often, tests of such systems amount to shallow tests of the user interface component because that's the only component that the tester can easily control.

---

[14] For example, the Java language encourages programmers to use the `toString` method to make internal objects printable. A programmer doesn't have to use it, since the debugger lets her see all the values in any object, but it simplifies debugging for objects she'll look at often. All testers need (roughly) is a way to call `toString` from some external interface.

[15] For a list of such commercial tools, see http://www.stlabs.com/marick/faqs/tools.htm. Follow the link to "Other Test Implementation Tools".

Finding failures is only the start

It's not enough to find a failure; you must also report it. Unfortunately, **poor bug reporting** is a classic mistake. Tester bug reports suffer from five major problems:

1. They do not describe how to reproduce the bug. Either no procedure is given, or the given procedure doesn't work. Either case will likely get the bug report shelved.

2. They don't explain what went wrong. At what point in the procedure does the bug occur? What should happen there? What actually happened?

3. They are not persuasive about the priority of the bug. Your job is to have the seriousness of the bug accurately assessed. There's a natural tendency for programmers and managers to rate bugs as less serious than they are. If you believe a bug is serious, explain why a <u>customer</u> would view it the way you do.[16] If you found the bug with an odd case, take the time to reproduce it with a more obviously common or compelling case.

4. They do not help the programmer in debugging. This is a simple cost/benefit tradeoff. A small amount of time spent simplifying the procedure for reproducing the bug or exploring the various ways it could occur may save a great deal of programmer time.

5. They are insulting, so they poison the relationship between developers and testers.

[Kaner93] has an excellent chapter (5) on how to write bug reports. Read it.

Not all bug reports come from testers. Some come from customers. When that happens, it's common for a tester to write a regression test that reproduces the bug in the broken version of the product. When the bug is fixed, that test is used to check that it was fixed correctly.

However, **adding only regression tests** is not enough. A customer bug report suggests two things:

1. That area of the product is buggy. It's well known that bugs tend to cluster.[17]

2. That area of the product was inadequately tested. Otherwise, why did the bug originally escape testing?

An appropriate response to several customer bug reports in an area is to schedule more thorough testing for that area. Begin by examining the current tests (if they're understandable) to determine their systematic weaknesses.

Finally, every bug report is a gift from a customer that tells you how to test better in the future. A common mistake is **failing to take notes for the next testing effort.**

---

[16] Cem Kaner suggests something even better: have the person whose budget will be directly affected explain why the bug is important. The customer service manager will speak more authoritatively about those installation bugs than you could.

[17] That's true even if the bug report is due to a customer misunderstanding. Perhaps this area of the product is just too hard to understand.

The next product will be somewhat like this one, the bugs will be somewhat like these, and the tests useful in finding those bugs will also be somewhat like the ones you just ran. Mental notes are easy to forget, and they're hard to hand to a new tester. Writing is a wonderful human invention: use it. Both [Kaner93] and [Marick95] describe formats for archiving test information, and both contain general-purpose examples.

Theme Five: Technology Run Rampant

Test automation is based on a simple economic proposition:

- If a manual test costs $X to run the first time, it will cost just about $X to run each time thereafter, whereas:

- If an automated test costs $Y to create, it will cost almost nothing to run from then on.

$Y is bigger than $X. I've heard estimates ranging from 3 to 30 times as big, with the most commonly cited number seeming to be 10. Suppose 10 is correct for your application and your automation tools. Then you should automate any test that will be run more than 10 times.

A classic mistake is to ignore these economics, **attempting to automate all tests**, even those that won't be run often enough to justify it. What tests clearly justify automation?

- Stress or load tests may be impossible to implement manually. Would you have a tester execute and check a function 1000 times? Are you going to sit 100 people down at 100 terminals?
- Nightly builds are becoming increasingly common. (See [McConnell96] or [Cusumano95] for descriptions of the procedure.) If you build the product nightly, you must have an automated "smoke test suite". Smoke tests are those that are run after every build to check for grievous errors.
- Configuration tests may be run on dozens of configurations.

The other kinds of tests are less clear-cut. Think hard about whether you'd rather have automated tests that are run often or ten times as many manual tests, each run once. Beware of irrational, emotional reasons for automating, such as testers who find programming automated tests more fun, a perception that automated tests will lead to higher status (everything else is "monkey testing"), or a fear of not rerunning a test that would have found a bug (thus leading you to automate it, leaving you without enough time to write a test that would have found a different bug).

You will likely end up in a compromise position, where you have:

1. a set of automated tests that are run often.

2. a well-documented set of manual tests. Subsets of these can be rerun as necessary. For example, when a critical area of the system has been extensively changed, you

might rerun its manual tests. You might run different samples of this suite after each major build. [18]

3. a set of undocumented tests that were run once (including exploratory "bug bash" tests).

Beware of **expecting to rerun all manual tests**. You will become bogged down rerunning tests with low bug-finding value, leaving yourself no time to create new tests. You will waste time documenting tests that don't need to be documented.

You could automate more tests if you could lower the cost of creating them. That's the promise of **using GUI capture/replay tools to reduce test creation cost**. The notion is that you simply execute a manual test, and the tool records what you do. When you manually check the correctness of a value, the tool remembers that correct value. You can then later play back the recording, and the tool will check whether all checked values are the same as the remembered values.

There are two variants of such tools. What I call the first generation tools capture raw mouse movements or keystrokes and take snapshots of the pixels on the screen. The second generation tools (often called "object oriented") reach into the program and manipulate underlying data structures (widgets or controls). [19]

First generation tools produce unmaintainable tests. Whenever the screen layout changes in the slightest way, the tests break. Mouse clicks are delivered to the wrong place, and snapshots fail in irrelevant ways that nevertheless have to be checked. Because screen layout changes are common, the constant manual updating of tests becomes insupportable.

Second generation tools are applicable only to tests where the underlying data structures are useful. For example, they rarely apply to a photograph editing tool, where you need to look at an actual image - at the actual bitmap. They also tend not to work with custom controls. Heavy users of capture/replay tools seem to spend an inordinate amount of time trying to get the tool to deal with the special features of their program - which raises the cost of test automation.

Second generation tools do not guarantee maintainability either. Suppose a radio button is changed to a pulldown list. All of the tests that use the old controls will now be broken.

GUI interface changes are of course common, especially between releases. Consider carefully whether an automated test that must be recaptured after GUI changes is worth having. Keep in mind that it can be hard to figure out what a captured test is attempting to accomplish unless it is separately documented.

---

[18] An additional benefit of automated tests is that they can be run faster than manual tests. That allows you to reduce the time between completion of a build and completion of its testing. That can be especially important in the final builds, if only to avoid pressure from executives itching to ship the product. You're trading fewer tests for faster time to market. That can be a reasonable tradeoff, but it doesn't affect the core of my argument, which is that not all tests should be automated.

[19] These are, in effect, another example of tools that add test support code to the program.

As a rule of thumb, it's dangerous to assume that an automated test will pay for itself this release, so your test must be able to survive a reasonable level of GUI change. I believe that capture/replay tests, of either generation, are rarely robust enough.

An alternative approach to capture/replay is scripting tests. (Most GUI capture/replay tools also allow scripting.) Some member of the testing team writes a "test API" (application programmer interface) that lets other members of the team express their tests in less GUI-dependent terms. Whereas a captured test might look like this:

```
text $main.accountField "12"
click $main.OK
menu $operations
menu $withdraw
click $withdrawDialog.all
...
```

a script might look like this:

```
select-account 12
withdraw all
...
```

The script commands are subroutines that perform the appropriate mouse clicks and key presses. If the API is well-designed, most GUI changes will require changes only to the implementation of functions like withdraw, not to all the tests that use them.[20] Please note that well-designed test APIs are as hard to write as any other good API. That is, they're hard, and you shouldn't expect to get it right the first time.

In a variant of this approach, the tests are data-driven. The tester provides a table describing key values. Some tool reads the table and converts it to the appropriate mouse clicks. The table is even less vulnerable to GUI changes because the sequence of operations has been abstracted away. It's also likely to be more understandable, especially to domain experts who are not programmers. See [Pettichord96] for an example of data-driven automated testing.

Note that these more abstract tests (whether scripted or data-driven) do not necessarily test the user interface thoroughly. If the Withdraw dialog can be reached via several routes (toolbar, menu item, hotkey), you don't know whether each route has been tried. You need a separate (most likely manual) effort to ensure that all the GUI components are connected correctly.

Whatever approach you take, don't fall into the trap of **expecting regression tests to find a high proportion of new bugs.** Regression tests discover that new or changed code breaks what used to work. While that happens more often than any of us

---

[20] The "Joe Gittano" stories and essays on my web page, http://www.stlabs.com/marick/root.htm, go into this approach in more detail.

would like, most bugs are in the product's new or intentionally changed behavior. Those bugs have to be caught by new tests.
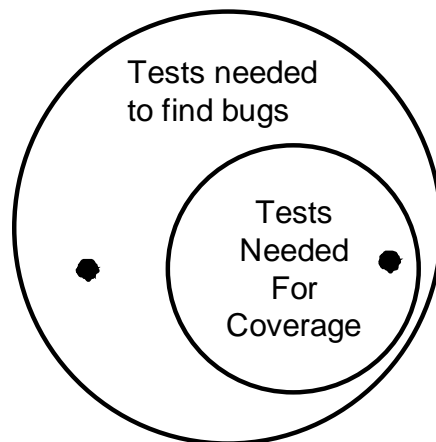
I © code coverage

GUI capture/replay testing is appealing because it's a quick fix for a difficult problem. Another class of tool has the same kind of attraction.

The difficult problem is that it's so hard to know if you're doing a good job testing. You only really find out once the product has shipped. Understandably, this makes managers uncomfortable. Sometimes you find them **embracing code coverage with the devotion that only simple numbers can inspire.** Testers sometimes also become enamored of coverage, though their romance tends to be less fervent and ends sooner.

What is code coverage? It is any of a number of measures of how thoroughly code is exercised. One common measure counts how many statements have been executed by any test. The appeal of such coverage is twofold:

1. If you've never exercised a line of code, you surely can't have found any of its bugs. So you should design tests to exercise every line of code.

2. Test suites are often too big, so you should throw out any test that doesn't add value. A test that adds no new coverage adds no value.

Only the first sentences in (1) and (2) are true. I'll illustrate with this picture, where the irregular splotches indicate bugs:

If you write only the tests needed to satisfy coverage, you'll find bugs. You're guaranteed to find the code that always fails, no matter how it's executed. But most bugs depend on how a line of code is executed. For example, code with an off-by-one error fails only when you exercise a boundary. Code with a divide-by-zero error fails only if you divide by zero. Coverage-adequate tests will find some of these bugs, by sheer dumb luck, but not enough of them. To find enough bugs, you have to write additional tests that "redundantly" execute the code.

For the same reason, **removing tests from a regression test suite just because they don't add coverage** is dangerous. The point is not to cover the code; it's to have tests that can discover enough of the bugs that are likely to be caused when the code is changed. Unless the tests are ineptly designed, removing tests will just remove power. If they are ineptly designed, using coverage converts a big and lousy test suite to a small and lousy test suite. That's progress, I suppose, but it's addressing the wrong problem.[21]

A grave danger of code coverage is that it is concrete, objective, and easy to measure. Many managers today are **using coverage as a performance goal for testers**. Unfortunately, a cardinal rule of management applies here: "Tell me how a person is evaluated, and I'll tell you how he behaves." If a person is evaluated by how much coverage is achieved in a given time (or in how little time it takes to reach a particular coverage goal), that person will tend to write tests to achieve high coverage in the fastest way possible. Unfortunately, that means shortchanging careful test design that targets bugs, and it certainly means avoiding in-depth, repetitive testing of "already covered" code.[22]

Using coverage as a test design technique works only when the testers are both designing poor tests <u>and</u> testing redundantly. They'd be better off at least targeting their poor tests at new areas of code. In more normal situations, coverage as a guide to design only decreases the value of the tests or puts testers under unproductive pressure to meet unhelpful goals.

Coverage does play a role in testing, not as a guide to test design, but as a rough evaluation of it. After you've run your tests, ask what their coverage is. If certain areas of the code have no or low coverage, you're sure to have tested them shallowly. If that wasn't intentional, you should improve the tests by rethinking their design. Coverage has told you <u>where</u> your tests are weak, but it's up to you to understand <u>how</u>.

You might not entirely ignore coverage. You might glance at the uncovered lines of code (possibly assisted by the programmer) to discover the kinds of tests you omitted. For example, you might scan the code to determine that you undertested a dialog box's error handling. Having done that, you step back and think of all the user errors the dialog box should handle, not how to provoke the error checks on line 343, 354, and 399. By rethinking design, you'll not only execute those lines, you might also discover that several other error checks are entirely missing. (Coverage can't tell you how well you would have exercised needed code that was left out of the program.)

---

[21] Not all regression test suites have the same goals. Smoke tests are intended to run fast and find grievous, obvious errors. A coverage-minimized test suite is entirely appropriate.

[22] In pathological cases, you'd never bother with user scenario testing, load testing, or configuration testing, none of which add much, if any, coverage to functional testing.

There are types of coverage that point more directly to design mistakes than statement coverage does (branch coverage, for example).[23] However, none - and not all of them put together - are so accurate that they can be used as test design techniques.

One final note: Romances with coverage don't seem to end with the former devotee wanting to be "just good friends". When, at the end of a year's use of coverage, it has not solved the testing problem, I find testing groups **abandoning coverage entirely.** That's a shame. When I test, I spend somewhat less than 5% of my time looking at coverage results, rethinking my test design, and writing some new tests to correct my mistakes. It's time well spent.

Acknowledgements

My discussions about testing with Cem Kaner have always been illuminating. The LAWST (Los Altos Workshop on Software Testing) participants said many interesting things about automated GUI testing. The LAWST participants were Chris Agruss, Tom Arnold, James Bach, Jim Brooks, Doug Hoffman, Cem Kaner, Brian Lawrence, Tom Lindemuth, Noel Nyman, Brett Pettichord, Drew Pritsker, and Melora Svoboda. Paul Czyzewski, Peggy Fouts, Cem Kaner, Eric Petersen, Joe Strazzere, Melora Svoboda, and Stephanie Young read an earlier draft.

References

[Cusumano95]
    M. Cusumano and R. Selby, *Microsoft Secrets,* Free Press, 1995.
[Dyer92]
    Michael Dyer, *The Cleanroom Approach to Quality Software Development,* Wiley, 1992.
[Friedman95]
    M. Friedman and J. Voas, *Software Assessment: Reliability, Safety, Testability,* Wiley, 1995.
[Kaner93]
    C. Kaner, J. Falk, and H.Q. Nguyen, *Testing Computer Software (2/e),* Van Nostrand Reinhold, 1993.
[Kaner96a]
    Cem Kaner, "Negotiating Testing Resources: A Collaborative Approach," a position paper for the panel session on "How to Save Time and Money in Testing", in *Proceedings of the Ninth International Quality Week* (Software Research, San Francisco, CA), 1996. (http://www.kaner.com/negotiate.htm)
[Kaner96b]
    Cem Kaner, "Software Negligence & Testing Coverage," in *Proceedings of STAR 96*, (Software Quality Engineering, Jacksonville, FL), 1996. (http://www.kaner.com/coverage.htm)

---

[23] See [Marick95], chapter 7, for a description of additional code coverage measures. See also [Kaner96b] for a list of more than one hundred types of coverage.

[Lyu96]
Michael R. Lyu (ed.), *Handbook of Software Reliability Engineering,* McGraw-Hill, 1996.

[Marick95]
Brian Marick, *The Craft of Software Testing,* Prentice Hall, 1995.

[Marick97]
Brian Marick, "The Test Manager at the Project Status Meeting," in *Proceedings of the Tenth International Quality Week* (Software Research, San Francisco, CA), 1997. (http://www.stlabs.com/~marick/root.htm)

[McConnell96]
Steve McConnell, *Rapid Development,* Microsoft Press, 1996.

[Moore91]
Geoffrey A. Moore, *Crossing the Chasm,* Harper Collins, 1991.

[Moore95]
Geoffrey A. Moore, *Inside the Tornado,* Harper Collins, 1995.

[Musa87]
J. Musa, A. Iannino, and K. Okumoto, *Software Reliability : Measurement, Prediction, Application*, McGraw-Hill, 1987.

[Nielsen93]
Jakob Nielsen, *Usability Engineering,* Academic Press, 1993.

[Pettichord96]
Bret Pettichord, "Success with Test Automation," in *Proceedings of the Ninth International Quality Week* (Software Research, San Francisco, CA), 1996. (http://www.io.com/~wazmo/succpap.htm)

[Rothman96]
Johanna Rothman, "Measurements to Reduce Risk in Product Ship Decisions," in *Proceedings of the Ninth International Quality Week* (Software Research, San Francisco, CA), 1996. (http://world.std.com/~jr/Papers/QW96.html)

[Voas91]
J. Voas, L. Morell, and K. Miller, "Predicting Where Faults Can Hide from Testing," *IEEE Software*, March, 1991.

Some Classic Testing Mistakes

The role of testing

- Thinking the testing team is responsible for assuring quality.
- Thinking that the purpose of testing is to find bugs.
- Not finding the important bugs.
- Not reporting usability problems.
- No focus on an estimate of quality (and on the quality of that estimate).
- Reporting bug data without putting it into context.
- Starting testing too late (bug detection, not bug reduction)

Planning the complete testing effort

- A testing effort biased toward functional testing.
- Underemphasizing configuration testing.
- Putting stress and load testing off to the last minute.
- Not testing the documentation
- Not testing installation procedures.
- An overreliance on beta testing.
- Finishing one testing task before moving on to the next.
- Failing to correctly identify risky areas.
- Sticking stubbornly to the test plan.

Personnel issues

- Using testing as a transitional job for new programmers.
- Recruiting testers from the ranks of failed programmers.
- Testers are not domain experts.
- Not seeking candidates from the customer service staff or technical writing staff.
- Insisting that testers be able to program.
- A testing team that lacks diversity.
- A physical separation between developers and testers.
- Believing that programmers can't test their own code.
- Programmers are neither trained nor motivated to test.

The tester at work

- Paying more attention to running tests than to designing them.
- Unreviewed test designs.
- Being too specific about test inputs and procedures.
- Not noticing and exploring "irrelevant" oddities.
- Checking that the product does what it's supposed to do, but not that it <u>doesn't</u> do what it <u>isn't</u> supposed to do.
- Test suites that are understandable only by their owners.

- Testing only through the user-visible interface.
- Poor bug reporting.
- Adding only regression tests when bugs are found.
- Failing to take notes for the next testing effort.

Test automation

- Attempting to automate all tests.
- Expecting to rerun manual tests.
- Using GUI capture/replay tools to reduce test creation cost.
- Expecting regression tests to find a high proportion of new bugs.

Code coverage

- Embracing code coverage with the devotion that only simple numbers can inspire.
- Removing tests from a regression test suite just because they don't add coverage.
- Using coverage as a performance goal for testers.
- Abandoning coverage entirely.