Test Planning for Consumer Software

Presented at the Quality Week Conference

San Francisco

May 12, 1988

Cem Kaner

Human Interface Technologies
801 Foster City Blvd. #101
Foster City, CA 94404.

Consumer software, such as word processors, games, and graphics programs, must be good, but is not life-critical – it doesn't have to be *perfect*. Accordingly, your testing budget is modest compared to those for life-critical software. So, you have to take different shortcuts from the ones discussed so far at this conference. My talk is about the focus and strategy of testing consumer software.

The talk's key points

1) The only reason to write a test plan is to help find bugs. Period. Anything else is a diversion of resources.

2) The approaches discussed in this conference work from the inside out, from the code to the world. Don't do this. Develop your test plan from the consumer's eye in.

3) Adopt an evolutionary test planning approach, rather than a waterfall approach.

1. Write test plans to find bugs

We write test plans for two very different types of reasons. Sometimes the test plan is a product; sometimes it's a tool. It's too easy, but also too expensive, to confuse these goals. The product is much more expensive than the tool.

The test plan as a product

Before some companies (e.g. AT&T) will market another party's (e.g. your) software-intense product, they scrutinize its software test plan. A well-organized, tidy, highly detailed test plan helps convince the customer (the remarketer) that the product was thoroughly tested and that, if they need to take over maintenance of the software (e.g. if you go bankrupt), they'll be able to rapidly figure out how to retest their fixes. To close the sale, you'd probably write whatever test plan they need, in whatever format, to whatever level of detail – as long as the they'll pay for it.

- When you sell software to the military, you sell them (and charge them for) Mil Spec test plans. Otherwise, they won't buy your code.
- If you develop a medical product that requires FDA inspection, you'll make the test plan impressive, in order to help the FDA feel warm and cozy.
- One more example: a software development house might try to leverage off the expertise of an independent test agency by having the agency develop the plan, which they (the developers) will execute without further agency help. The agency must write a document that is very organized and detailed, or the developers won't know how to execute it.

Each of the above test plans is useful for finding bugs. However, it's important to note that in each case, if you could find more bugs in the time available by spending more time thinking and testing and less time writing an impressively formatted test plan, you would still opt for the fancy document (test plan) because the customer wants it.

The test plan as a tool

If you don't care how pretty the test plan is as long as it helps you find the most bugs in the least time, you're talking about using the test plan as a tool, This is what we should do in testing consumer software.

When I go through ANSI/IEEE standards on test plan documentation, I see requests for test design specifications, and test case specifications, and test logs, and test-various-identifiers, and test procedure specifications, and test item transmittal reports, and introductions, and input/output specifications, and special procedure requirements specifications, and intercase dependency notes, and test deliverables lists, and schedules, and staff plans, and responsibilities per staffer, and test suspension and resumption criteria, and masses of other paper.

Listen carefully when people tell you that standards help you generate the masses of paper more quickly. They do, but so what? The question is, how much of this quickly generated paper helps us find more bugs more quickly?

Whenever I've clocked it, it's taken vast time to do any significant proportion of this. It seems to me that when I've tried to develop the books of test plans that "professional standards" call for, I end up with more *stuff* than the IRS asks for in an audit. And I sit back and say "why am I doing this? Why bury myself in paper?" Because what I'm doing is *creating* paper. I'm not finding bugs. And the folks that buy my product are asking for something that makes the right sounds, that draws the right pictures, that types the text in the right places at the right times. They don't care how it was tested. They just care that it works.

So, the criterion that I want to impose on test planning is purely this: a test plan is valuable if it clarifies your thinking about potential bugs, including telling you about the scope of your own testing. It's feedback to you, It's a way of organizing. Beyond that, it's just taxes. And my approach to taxes in this business is to evade them.

2. Take the customer's view when looking for bugs

Over the last few days, you've *studied* path and branch testing and other methods of finding bugs by analyzing the *code*. This glass box testing is important work, but it will miss many problems. I'll shortly recommend that, unless you have lots of extra testing time, you should leave glass-box to the programmers (even if they don't do as much as you'd like). Instead, you should test the running code, from the outside, working and stressing it in all the many ways that your customers might.

But before talking about the most efficient ways to find problems, I have to make *sure* we're using the same terminology, so here *goes.*

- *Quality*: Some people define quality operationally, in terms of a match to a specification. That's no good for consumer software. If we perfectly implement a specification of a lousy product, the result will be a low quality product that won't sell. Instead, let's say that a program is good if people who want to use it are happy with it, and it's bad if they aren't. It's bad if it doesn't do what they want, if it doesn't provide the functions they need, if it's unreliable, or if it is clumsy or annoying to use.
- Bug: A program has a bug when it fails to do what its end customer reasonably expects it to do. (Myers, The Art of Software Testing, Wiley, 1979).
- *Coding error*: the program's behavior doesn't match what the programmer intended to happen.
- *Design* error: the program works the way the programmer says it should, but it doesn't do what the user reasonably wants or expects. "Wants" is a funny word because it raises marketing issues that I want to avoid. But if a customer expects to be able to do something with a product and s/he can't, or s/he finds it very clumsy, or s/he gets the wrong answers, s/he won't be happy with the program. It's vital to look at the program from the perspective: How *will this go over in the world?*

What path testing misses

As you've been taught over the last two days, it is important for someone to test paths and branches directly from the code. But this doesn't tell us the whole story. We are kidding ourselves when we say "100% testing coverage," meaning only that we've checked all branches. At this "100% coverage" level, we'd be lucky to have found half the bugs.

Path and branch testing is relatively easy and straightforward, and most programmers do it. Modern tools, like the ones discussed over the last couple of days, make it even easier for programmers to do this type of testing themselves. As testers we should look for the other problems, the ones the programmers won't find by doing the code-driven testing that they so naturally do.

Here are three examples of bugs in MS-DOS systems that would not be detected by path and branch tests.

1) Dig up some early (pre-1984) PC programs. Hit the space bar while you boot the program. In surprisingly many cases, you'll have to turn off the computer because interrupts weren't disabled during the disk I/O. The interrupt is clearly an unexpected event, so no branch in the code was written to cope with it. You won't find the absence of a needed branch by testing the branches that are there.

2) Attach a color monitor and a monochrome monitor to the same PC and try running some of the early PC games. In the dual monitor configuration, many of these destroy the monochrome monitor (smoke, mess, a spectacular bug).

3) Connect a printer to a PC, turn it on, and switch it off line. Now have a program try to print to it. If the program doesn't hang this time, try again with a different version of MS-DOS (different release number or one slightly customized for a particular computer), Programs (the identical code, same paths, same branches) often crash when tested on configurations other than those the programmer(s) used for development.

It's hard to find these bugs because they aren't anticipated in the code. There are no paths and branches for them. You won't find them by executing every line in the code. You won't find them until you step away from the code, look at the program from the outside, and ask how customers will use the program, on what types of equipment.

Testers are too often encouraged to think that they're not doing "real" testing unless they do path and branch testing, or other glass-box testing. I say that's the wrong direction for a test group. Yes, of course, someone should do path testing. And if programmers won't, someone (maybe a tester) should join the programming team to do it. But look at Table 1 for problems you won't find during path testing. As the tester, you're the one who has to find these. To have time to do that, you have to let someone else worry about code paths. You have to look at the program from the outside, from the viewpoint of your customers, to understand who it can fail as they use it.

WHAT CODE PATHS DON'T TELL YOU

---

1. Timing-related bugs
2. Unanticipated error conditions
3. Special data conditions
4. Validity of displayed stuff
5. User interface consistency
6. User interface everything-else
7. Interaction with background tasks
8. Configuration / compatibility
9. Volume, load, hardware fault

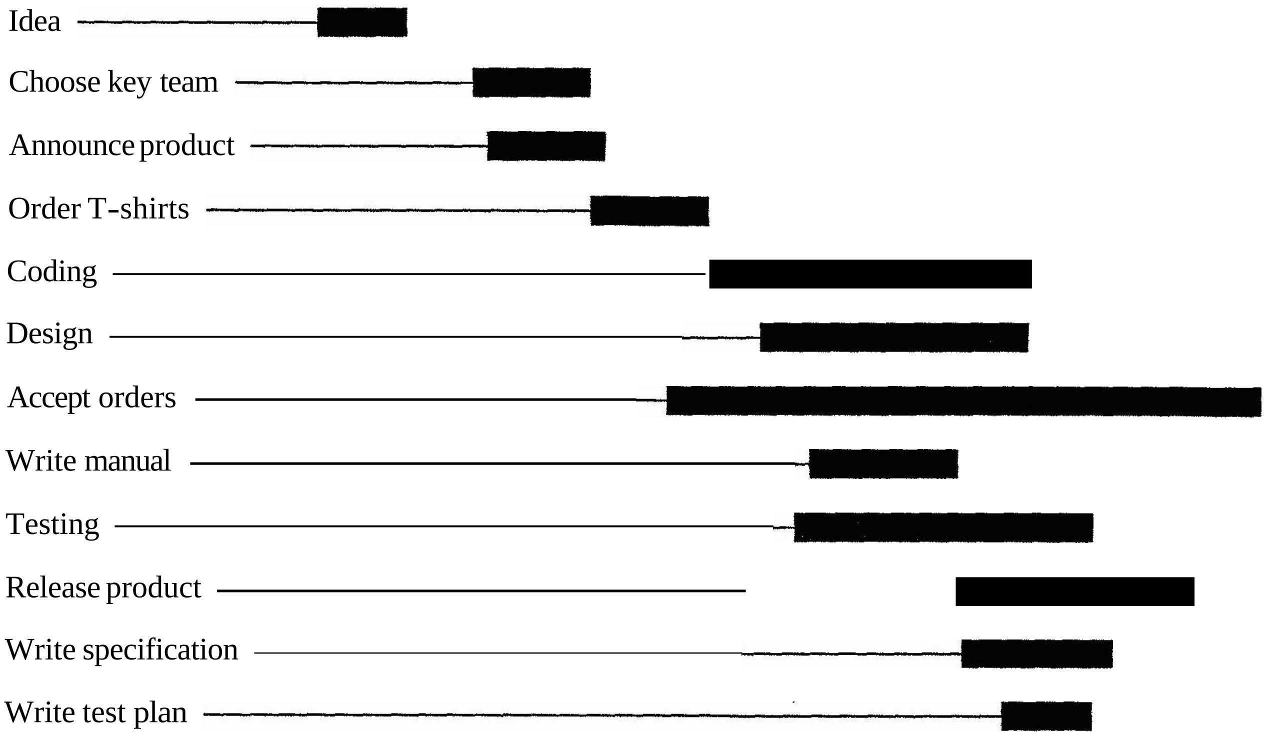
Slide 1 lists problems that path and branch testing can easily miss. Here are some notes:

- ***Timing related problems***: As a program moves from state to state, look for a way to insert an unexpected event into the transition period. This is a window of opportunity to find a bug. Widen it by testing on a slower processor or on a multitasking system that is running many concurrent tasks.
- ***Error conditions***: In my experience, programs are at their most vulnerable when reacting to a user error or a device error. You'll often be surprised at how many oddities you find when you check a program's response to each possibly keystroke when a dialog box is up alerting you to an error.
- ***Data conditions***: How does the program handle division by zero? If there is no test for this in the program, how will you catch it with branch testing?
- ***Display validity:*** Many tests verify that the program displays a string as it progresses down a code path, but how do you know that it is displaying the right string?
- ***User interface***: We think of word processors as a tool for improving efficiency, but some are so badly designed that a fast typist takes four times as long to type a letter on the word processor than on a typewriter. Perfect implementations of bad designs yield bad products.

3. Adopt an evolutionary approach to test plan development

Traditional software development books say that "real development teams" follow the waterfall method. Under the waterfall, one works in phases, from requirements analysis to various types of design and specification, to coding, final testing, and release.

The Product Development Cycle



In software design and development as a whole, there are very serious problems with the waterfall method. For details, see Tom Gilb's excellent book *(Principles of Software Engineering Management, Addison-Wesley, 1988),* and his references.. In consumer software, there's a further pragmatic problem: most of us don't and won't use the waterfall. The approach we take is often caricatured by charts like Slide 2.

Some testers make a mission of exhorting consumer software developers to follow the waterfall, but all they achieve is a loss of credibility.

As testers, we force reality down a lot of peoples' throats. We have to do the same for ourselves. It is more important to figure out how to test well in the context of what is done than to develop models and procedures for testing in the context of what is not done, and will not be done.

Apart from pragmatics, I think a strict waterfall approach is wrong for testing. In waterfall-based testing, you do your thinking and test planning early and you execute the tests later. But as organized as this looks on paper, you actually learn the most about the product and how to make it fail when you test it. Do you really want to schedule the bulk of thinking before the bulk of your learning?

There's a joke about consumer software that products aren't released – they escape!

The challenge for m is to make sure that we've always tested a product as well as possible in the time available, became we never know when a product will escape.

In software development, Gilb (1988) says deliver a small piece, test it, get to like it eventually, then add another small piece that adds significant functionality. Test that as a system. Then add the next piece and see what it does to the system. Note how much low-cost opportunity you have to reappraise requirements and refine the design as you understand the application better. Also, note that you are constantly delivering a working, useful product. If you add functionality in priority order, you could stop development at any time and know that the most important work has been done. Over time, the product evolves into a very rich, reliable, useful product. This is the evolutionary method.

In testing and especially in test planning, we can also be evolutionary, whether or not the program was developed in an evolutionary way. Rather than trying to develop one huge test plan, start small. Build a piece of what will become part of the large, final test plan, and use it to find bugs, Add new sections to the test plan, or go into depth in new areas, and use each one. Develop new sections in priority order, so that on the day the executives declare an end to testing and ship the product (an event that could happen at any time), you'll know that you've run the best test set in the time available.

Tactics of Evolution (1)

* * Start Broad * *

---

1. Full review of (user) documentation
2. Superficial function list
3. Analyze inputs, limits, ignoring most interactions

My approach requires parallel work on testing and on the test plan. You never let one get far ahead of the other. When you set aside a day for test planning, leave an hour or two to try out your ideas at the keyboard. When you focus on test execution, keep a notepad handy for recording new ideas for the test plan. You will eventually get an excellent test plan, because you've preserved your best creative ideas. But it starts out sketchy. It is fleshed out over time. In the meantime, you test a lot, find lots of bugs, and learn a lot about the program.

The order of development of the test plan

Slide 3 describes the first steps for developing the test plan. Start by going through the entire program at a superficial level. Try to maintain a uniform level of coverage, superficial coverage, across the whole program. Find out what people will find in the first two hours of use, and get this out of the program early.

- ***Test** against the documentation*: Start by comparing the program's behavior and whatever draft of the user documentation you get. If you also have a specification, test against that too. Compare the manual and the product line by line and keystroke by keystroke. You'll find plenty of problems and provide lots of help to the programmers and the manual writers.

  Begin creating documentation that's organized for efficient testing, such as a function list. Such a list includes everything the program's supposed to be able to do. Make the list, and try everything out. Your list won't be complete at first – there will be undocumented features, and it will lack depth -- but it'll grow into a complete list over time. I'll discuss the gradual refinement of the function list later.

- ***Do a simple analysis of limits*** **Try** reasonable limits everywhere that you can enter data. If the program doesn't crash, try broader limits. Draft user manuals rarely indicate boundary conditions. Specifications (if you have such things) ***too*** often describe what was planned before the developers started writing the code and changed everything. In your testing, you'll find out what the real limits are. Write them down. Then circulate your notes for the programmers and writers to look at, use, and add to.

In sum, start by building a foundation. Use an outline processor so you can reorganize and restructure the foundation easily. In laying the foundation, you test the whole program, albeit not very thoroughly, This lets you catch the most obvious problems right away, and get them fixed. And as you add depth, you are adding detail to a centrally organized product.

Where to focus next, where to add depth

Once you finish the first superficial scan of the program, what do you do next. What are the most important areas to test? What's the best area of focus? There's no magic formula. It depends on what you know **and** what your instincts suggest will be most fruitful this time, but it will probably be in one of the six areas listed in Slide 4.

Tactics of Evolution (2)

* * Targets for focus * *

1, Most likely errors
2. Most visible errors
3. Most often used program areas
4. Distinguishing areas of the program
5. Hardest areas to **fix**
6. Most understood by you

Most likely errors: If you know where there are lots of bugs, go there first and report them.

Most visible errors: Alternatively, start where customers will look first and hardest. Look in the most often used program areas, the most publicized areas, and the places that really make your program distinct from the others, or make it critically functional for the user. Features that are nice to have but you can live without are tested later. If they don't work, that's bad. But it's worse if the core functionality doesn't work.

- ***Distinguishing area* of *the* program:** If you're selling a database and you claim that it sorts 48 times faster than your competitor, you better test sorting because that's why people are buying your program. If your sorting is very fast but it doesn't work, customers will get grumpy. It's important to do early testing on heavily optimized areas that distinguish your program because heavily optimized code is often hard to fix. You want to report these bugs early to give the programmers a fighting chance to fix them.

- Hardest areas ***to fix***: Sit with the programmer and ask, "suppose the worst things happened to you. If I found bugs in the most horrible areas that you don't ever want to think about, what areas would those be?" Some programmers will tell you. Go right to those areas and beat on them. Do it now, when it's four months before the program will ship, to give the staff a chance to fix what you find. If you find these bugs a week before the scheduled ship date, the programmer will have a heart attack or quit and you'll never get them fixed.

- ***Most understood* by *you:*** Maybe you've read the code or you understand something about applications of this kind. Here's an area you understand, that you can test well right away. As to the rest, you're learning how to test the program as you test. If you're an expert in one area, test it first and test how it interacts with the other areas. Even if it's not a critical area, you'll gain good experience with the program and find bugs too. This will be a base: it will help you go much more effectively, and much more quickly, into the next area.

What should we cover in a test plan?

Slide 5 lists some of the areas covered in a good test plan or, more likely, in a good group of test plans. There's no need to put all of these areas into one document.

Some Areas to Cover in a Test Plan

1. Acceptance test (into testing)
2. Control flow
3. Data
4. Configuration / compatibility
5. Stress tests
6. User interface
7. Regression
8. Performance
9. Potential bugs
10. Beta tests
11. Release tests

Discussing these test areas in detail is almost embarrassing in this talk because you already know so much about them. But here are a few notes,

The most important initial note is to look at these areas with a sense of perspective. Good programmers are responsible people. They did lots of testing. They just didn't do the testing you're going to do. If you make your testing orthogonal to what they did, you'll find things that they didn't find. If you come up with tests from the outside that they wouldn't have come up with, you can win.

- ***Acceptance rests (into testing):*** When dealing with lots of project managers who are competing to pump products through your test group, you need acceptance tests. The problem is that they have an incentive to get their code into your group, and lock up your resources, as soon as possible. On the other hand, if you're tight on staff, you have to push back and insist that the program be reasonably stable before you can commit staff to it. Publish acceptance tests for each programs. Be so clear about your criteria that the programmers can run the test themselves and know whether they'll pass it before they submit the program to you. Many project managers will run the test (especially if they understand that you'll kick the program out of testing if it doesn't pass the test), and will make sure that the product's most obvious bugs are fixed before you ever see it.

  This brief test covers only the essential behavior of the program. It should last a few hours, a few days at most in a particularly complex system. Also, it is often a candidate for automation.

  Control flow: When you ask about control flow, you're asking how you can get the program from one state to another. You're going to test the visible control Bow, rather than the functional, internal flow- What are the diierent ways that you can get to a dialog box? What different menu paths can you take to get to the printer? What parameters can you give with commands to force the program into other states?

- Data: there are all sorts of ways to test how data flows through a system. At this conference, you probably know more about data testing than I do, so I won't embarrass myself with that.

- ***Compatibility/configuration tests:*** Everybody's sick of thinking about hardware compatibility, so I'll let it alone here. But don't forget that you have to work with other software, running in the foreground, running as terminate and stay resident, or running in the background. Maybe you have to read and write data in compatible formats. How do you operate with all those formats? It's not just hardware compatibility. When was the last time you checked out what other software your program is compatible with? Testers often don't check software compatibility as carefully as magazines like InfoWorld. The consequences are embarrassing.

- ***Stress tests:*** we know enough about these that I'll skip discussing them.

- ***User interface tests:*** if the program seems inconsistent, if it confuses you, you can bet that some customers will have the same problem. Report a bug when you find that you keep tripping over the same area of the program or that the documentation of an area is hard to understand because the program confused the writer, or it has too many special cases. It doesn't matter if the program works to spec because customers still won't like it.

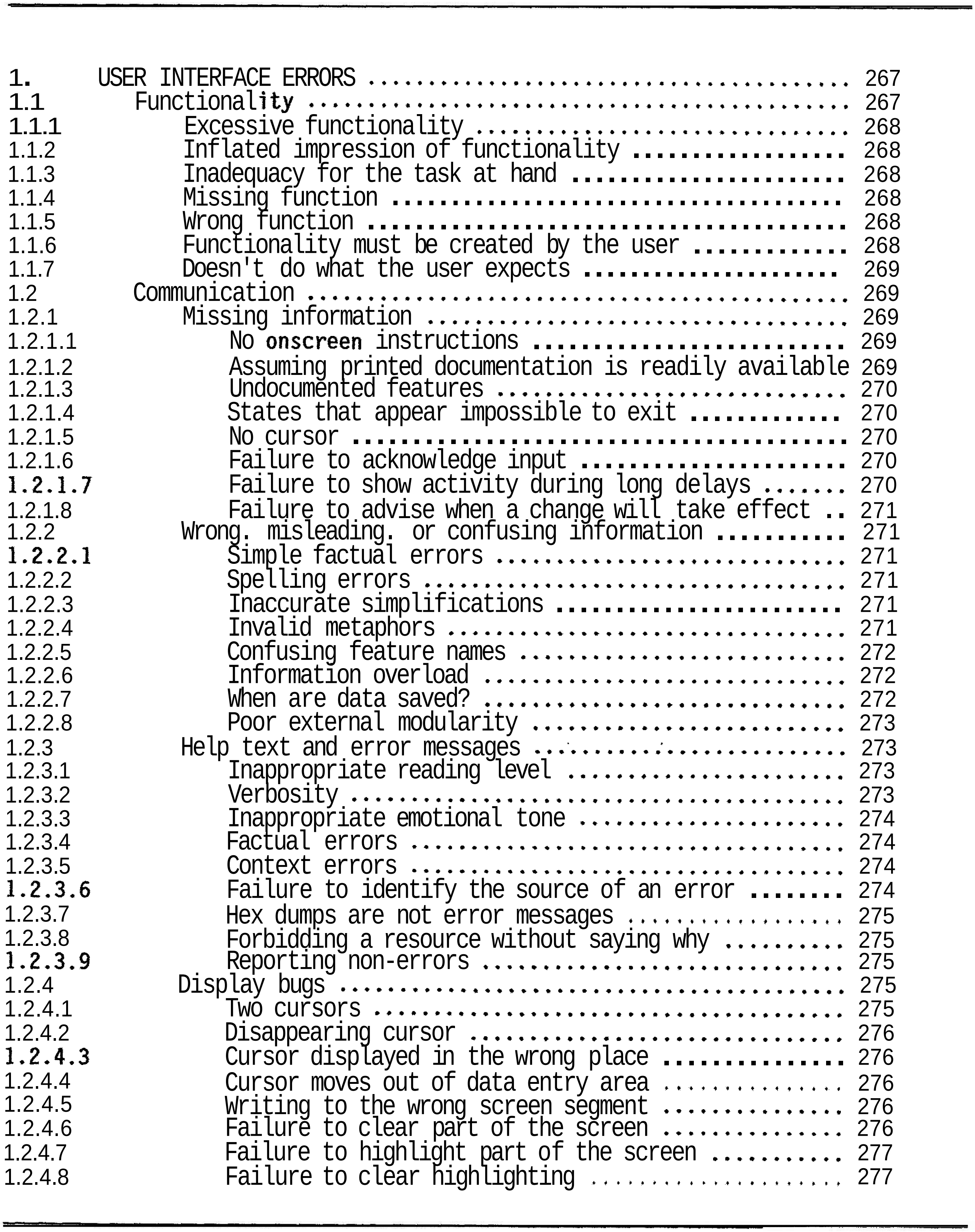
  You can do formal usability tests. You should also monitor what you're doing yourself. What kinds of user errors do you making while you use or test the code? Will you recognize it if an area of the program is leading you into a higher error rate?

- ***Performance:*** Some serious internal bugs show up most strongly as changes in the speed with which a task now gets done. Big changes for better or worse are suspect.

- ***Potential bugs:*** People rarely cover predictable bugs in an organized way. I have a list of about 550 of the kinds of bugs that I find in programs. When I find a bug, I try to think of a way to think of it in a little more general way. If I can extend my concept of the bug so that it isn't restricted to this program on this kind of machine, then it goes into my list.

  When I start wondering if a test plan is adequate, I use the list. I pick out a bug and say, "Here's one that ***could*** be in the program. If it ***could*** be in the program, whether it is or not, one of my tests better make sure that it's not in the program. So I can go back through the test plan and ask, "would I have found this bug if it's there?" I don't know if it's there, but if it is, would I have found it? If so, I'm getting reasonable coverage. If not, I just found what might be a big hole in the test plan.

The following page is from my book. *Testing Computer Software*. It's an example of the kind of list I glean ideas from. In the book I also have a long appendix that describes each of these and many other bugs in more detail.



An area in any test plan of mine is a list of bugs that I think might occur in this kind of a program. Sometimes I go through it formally, sometimes I just work with the program and say "Yeah, this or this or this or this could happen". Later, as I evolve the test plan, I can move these bugs into the control flow section or the data section. But right now, I can just put them into this bug list section and say, "these are interesting bugs. Let's look for them."

- Release *testing*: this is another area we all know lots about so I won't talk about it.

Beta *testing:* This is an area of testing that is less well understood and more controversial than the others so far.

We need feedback from customers before shipping a product. But we often try to get too much from too few people at the wrong times, using the wrong type of test. The problem is that there are probably are probably seven distinct classes of end user tests that we call beta tests.

Beta tests

1. Expert consulting
2. (Marketing) Testimonial / magazine reviews
3. (Marketing) Profile customer uses
4. Polish the design
5. Find bugs
6. Check performance or compatibility with specific equipment
7. Feature feedback for next release

- ***Expert consulting:*** early in development, marketing or development may talk with experts about the product vision and perhaps about a functional prototype. The goal is to determine how they like the overall product concept, what they think it needs, and what changes will make it more usable or competitive.

  Some companies get caught up in an idea that they shouldn't show outsiders anything until "beta", some late stage in development. After beta, the experts are consulted. By then it's too late to make the kinds of fundamental changes they request, so everybody gets frustrated.

  If you're going to use experts, use them early.

- ***Magazine reviewers:*** some reviewers love to suggest changes and save their best reviews for products they were successful in changing. To them, you have to send early copies of the program. To others, who want to evaluate final product without changing it, you want to send very late copies. You won't expect feedback from them, apart from last-minute bug discoveries, and no one should expect the programmers to make late design changes in response to design feedback from these late version reviewers. There's no time in the schedule to even evaluate their design feedback.

  The marketing department must decide, on a case by case basis, who gets early code and who gets it late.

- ***Testimonials*** might also be important for advertising. Again, marketing manages the flow of product to these people. Some get code early and get to feel that they contributed to the design. Others get almost-final code and can't contribute to the design.

- *Profiling customer uses and polishing the design:* it might be important to put almost-final product in the hands of representative customers and see how they actually use it. Their experience might influence the positioning of the product in initial advertising. Or their feedback might be needed to seek and smooth out rough edges in the product's design. To be of value, this type of test might leave preliminary product in customer hands for a month or more, to let them gain experience with the program. To allow time for polish to be implemented, in response to these customer results, you might need another month (or more).

  People often say that they do beta testing to find out how customers will use the product and to respond to the problems these sample customers raise. If you want any hope of success of this type of testing, budget at least 8 and preferably 10 or more weeks between the start of this testing and the release of final product to manufacturing.

The common problem with beta testing is that the test planners don't think through their objectives precisely enough. What is the point of running the test if you won't have time to respond to what you learn? What types of information do you expect from this test and why can't you get them just as well from in-house testing? How will you know whether you these outsiders have done the testing you wanted them to do?

Test planning support documents

This final area of the talk looks at some of the documents we can generate while we're testing. Again, the point of these documents is to help the testing at hand, not to look pretty later.

Examples of Test Planning Support Documents

---

1. Function list
2. Boundary chart
3. Keyboard convention chart or matrix
4. Input combination charts:
   List independent data items
5. Hardware compatibility chart
6. Hardware/software configuration test matrix
7. Decision tables
8. Bill of materials and marketing support docs

- ***Function list:*** evolve this. Start by listing all the user-visible functions. Include commands, menu choices, pulldowns, compiler options, and any other significant capabilities that you know are present in the program. This is your list of toplevel functions. Then ask what are the subfunctions and the subsubfunctions. Eventually you get a very detailed, structured listing of the program's capabilities. I use an outline processor to manage it. At any point in time, the outline is less detailed than you might like, but if you continue adding detail as you test and learn, the outline becomes an invaluable map of your knowledge of the program.

Evolving the Function List

1. List the top level, user-visible functions
   (commands, actions, menu options)
2. Deepen the list with subfunctions
   fall available options or menu choices from a main function.)
3. Show subfunctions to their deepest level
   (Each line at this level represents a fully parameterized choice -- something that will actually be executed.)
4. List entry and exit conditions for each function and subfunction
5. List all keyboard and other input device effects on dialogues in this function

Eventually, make each line a test case

You can take the list to a stage where each lower level line in the outline represents a function that is so well specified that if you do that operation, the computer will do exactly one thing. This is a test case. Expected results belong here -- you should write them down when you find time. Finally, analyze the list one step further: look at the entry and exit conditions to get into and out of this program function. Are there more ways to get into here than the ones listed -- often there are, often undocumented. Are there either error or normal ways to get out of this function at this point? How to abort? If you can't, that's a bug. Thinking about the possible ways into and out of each functions will expose bugs that you'll never find by looking inside the code.

- *Boundary charts*: In a boundary chart, you just list all the input data, all the types of input data, the variables being asked for, and the limits on them. If you know about a transition, a variable that will be used a different way depending on some internal value, list that too.

  We know how to do boundary charts. If you're not sure, read Myers. An important lesson is to never try to do the whole chart up front. First, it's always wrong. Second, it will take a week or three or more putting together a chart like this, and if that's what you're doing, you're not finding *any* bugs. It's critical to the project to find bugs early. The test plan details can wait on the bugs.

  To develop a boundary chart as I test, I start by identifying every place where I can enter data. This gives me a place where I can write test information as I think of it. That's already a big step beyond many test plans that I've seen.

Evolving the Boundary Chart

1. Identify all upper, lower, and intermediate transition data limits for each data entry field or requestor.
2. What does (should) the program do in the neighbourhood of each limit?
3. Make notes on limit-affecting interactions with other variables and/or with the place in the program that requests a change to this variable.

Start by listing all the entry fields and identify their functions. Assign limit values and further information as you learn more, but let yourself experiement with the variables from a full list from as early a time as you can.

Combinations of inputs. Again, the key is to build up information over time.

Whenever I think of how inputs can be combined, I'm appalled. I see all these variables that can combine in nine trillion zillion ways and I say that I can't deal with all this, so I pull out my random number generator to sample from them and it breaks.... It's too much. The result is that most folks that I know say, "we can't test these combinations anyway, so why bother?" They don't do any combination testing except what they do accidentally.

Evolving the Input Combinations Chart

1. List what you know as you learn it
2. Include positive information (this datum has the following effect on that variable.)
3. Include negative information (these two variables are independent).

We learn a lot about input combinations as we test. We learn about natural combinations of these variables, and about variables that seem totally independent- Go to the programmers with the list of variables and ask which ones are supposed to be totally independent. Don't expect their memories to be perfect -- check a few combinations of allegedly independent variables just to reassure yourself, but concentrate your time on the variables you expect trouble with.

That's all I have time for today, so I'll stop. Enjoy the rest of the conference