Improve the Power of Your Tests with Risk-Based Test Design

Cem Kaner, J.D., Ph.D.
Professor of Software Engineering
Florida Institute of Technology

Conference Abstract

Risk-based test management evaluates each area of a product and allocates higher testing budgets for areas of greater risk. Once you have the budget, how should you spend it?

Risk-based test design, on the other hand, is based on the idea that every test presents the program with an opportunity to fail. The first core task of risk-based test design is to imagine ways the program can fail. The second task is to design tests that are effective triggers for those failures. The most powerful tests are the ones that maximize a program's opportunity to fail.

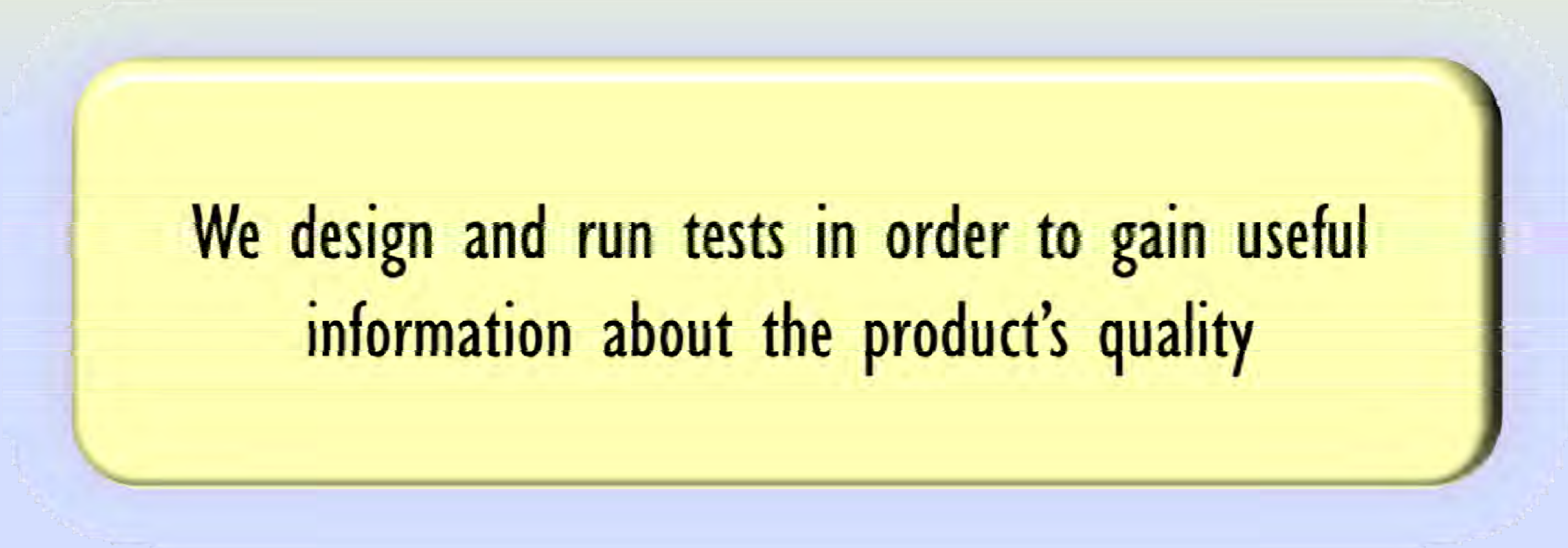
In this keynote, Cem will survey techniques for stretching your failure-related imagination, such as using guideword heuristics (as is commonly done in Failure Mode and Effects Analysis, for example) to quickly gain and apply knowledge about:

- The type of application.
- The environment and programming language.
- The project's management, development, and support history.

Cem will also examine ways of turning ideas about potential failure into tests, ranging from quicktests (straightforward applications of a theory of error, such as Whittaker's standard attacks) through tests that are more tightly customized to the specific concern. Join Cem and learn how to plan to "make it fail.

Preliminaries on Test design: What is testing?

Software testing is an empirical, technical investigation conducted to provide stakeholders with information about the quality of the product or service under test.



We design and run tests in order to gain useful information about the product's quality

Preliminaries on test design: Test and test case

Think of **a test** as a question that you ask the program.

- You run the test (the experiment) in order to answer the question.

A **test case** is a test

- Usually, when we just say “a test”, we mean something we do,
- Usually, when we say “test case,” we mean something that we have described / documented.

A **test idea** is the thought that guides our creation of a test. For example, “what’s the boundary of this variable? Can we test it?” is a test idea.

For our purposes today, the distinction between test and test case is irrelevant, and I will switch freely between the two terms.

Preliminaries on test design: Testing strategy

Given an information objective

- » My client wants me to find the most bugs
- » My client wants to know if the program meets the specification

The **testing strategy** specifies an integrated view of such things as:

- The techniques we'll rely on to help us generate tests that are best suited to giving us the type of information we need
- The logistical support (resources needed and available, at different times in the project)
- The human support (staff and their skills, other sources of information, people who will help you get the resources / staff you need, etc.

Preliminaries on test design: Information objectives

Find important bugs, to get them fixed

Assess the quality of the product

Help managers make release decisions

Block premature product releases

Help predict and control costs of product support

Check interoperability with other products

Find safe scenarios for use of the product

Assess conformance to specifications

Certify the product meets a particular standard

Ensure the testing process meets accountability standards

Minimize the risk of safety-related lawsuits

Help clients improve product quality & testability

Help clients improve their processes

Evaluate the product for a third party

Different objectives drive you toward different:

- Testing techniques
- Project mgmt styles
- Results reporting methods
- Politics on the project

Preliminaries on test design: : Test techniques

A test technique is essentially a recipe, or a model, that guides us in creating specific tests. Examples of common test techniques:

- Function testing
- Specification-based testing
- Domain testing
- Risk-based testing
- Scenario testing
- Regression testing
- Stress testing
- User testing
- All-pairs combination testing
- Data flow testing
- Build verification testing
- State-model based testing
- High volume automated testing
- Printer compatibility testing
- Testing to maximize statement and branch coverage

Test design: Examples of test techniques

- **Scenario testing**
  - Tests are complex stories that capture how the program will be used in real-life situations.
- **Specification-based testing**
  - Check every claim made in the reference document (such as, a contract specification). Test to the extent that you have proved the claim true or false.
- **Risk-based testing**
  - A program is a collection of opportunities for things to go wrong. For each way that you can imagine the program failing, design tests to determine whether the program actually will fail in that way.

Test design: Techniques differ in how to define a good test

Power. When a problem exists, the test will reveal it

Valid. When the test reveals a problem, it is a genuine problem

Value. Reveals things your clients want to know about the product or project

Credible. Client will believe that people will do the things done in this test

Representative of events most likely to be encountered by the user

Non-redundant. This test represents a larger group that address the same risk

Motivating. Your client will want to fix the problem exposed by this test

Maintainable. Easy to revise in the face of product changes

Repeatable. Easy and inexpensive to reuse the test.

Performable. Can do the test as designed

Refutability: Designed to challenge basic or critical assumptions (e.g. your theory of the user's goals is all wrong)

Coverage. Part of a collection of tests that together address a class of issues

Easy to evaluate.

Supports troubleshooting. Provides useful information for the debugging programmer

Appropriately complex. As a program gets more stable, use more complex tests

Accountable. You can explain, justify, and prove you ran it

Cost. Includes time and effort, as well as direct costs

Opportunity Cost. Developing and performing this test prevents you from doing other work

Test design: Techniques convey vision of a well-designed test

- **Scenario testing:**
- complex stories that capture how the program will be used in real-life situations
  - Good scenarios focus on validity, complexity, credibility, motivational effect
  - The scenario designer might care less about power, maintainability, coverage, reusability
- **Risk-based testing:**
- Imagine how the program could fail, and try to get it to fail that way
  - Good risk-based tests are powerful, valid, non-redundant, and aim at high-stakes issues (refutability)
  - The risk-based tester might not care as much about credibility, representativeness, performability—we can work on these after (if) a test exposes a bug

Preliminaries on test design: Tying this together

Design: "to create, fashion, execute, or construct according to plan; to conceive and plan out in the mind" (Websters)

- – Designing is not scripting. The representation of a plan is not the plan.

Usually (or at least, preferably) within the context of a **testing strategy**,

- the **test designer:**
  - – uses the **test ideas** / guidance contained in a **test technique**
  - – to craft a **specific test**
  - – that helps her collect a specific type of information (answer a reasonably specific question)

Risk-Based Design

- We often go **from technique to test**
  - Find all variables, domain test each
  - Find all spec paragraphs, make a relevant test for each
  - Find all lines of code, make a set of tests that collectively includes each
- It is much harder to go **from a failure mode to a test**
  - The program will crash?
  - The program will have a wild pointer?
  - The program will have a memory leak?
  - The program will be hard to use?
  - The program will corrupt its database?

How do we map from a failure mode to a test?

Design: Mapping from the failure mode to the test

- Imagine that someone called your company’s help desk and complained that the program had failed.
  - They were working in this part of the program
  - And the program displayed some junk on the screen and then crashed
  - They don’t know how to recreate the bug but that’s no surprise because they have no testing experience.

How would you troubleshoot this report?

In terms of the skills you use, working from a failure mode to a test is like trying to replicate someone else’s inadequate report of that failure.

Design: Mapping from the test idea to the test

- Let's create a slightly more concrete version of this example
  - Joe bought a smart refrigerator that tracks items stored in the fridge and prints out grocery shopping lists.
  - One day, Joe asked for a shopping list for his usual meals in their usual quantities.
  - The fridge crashed with an unintelligible error message.
- So, how to troubleshoot this problem?
- **First question: What about this error message?**
  - System-level (probably part of the crash, the programmers won't have useful info for us)
  - Application-level (what messages are possible at this point?)
  - **This leads us to our first series of tests:** Try to recreate every error message that can come from requesting a shopping list. Does this testing suggest anything?

Design: Evolving the test case from the story

- **Second question:** What makes a system crash?
  - Data overflow (too much stuff in the fridge?)
  - Wild pointer ("grunge" accumulates because we've used the fridge too long without rebooting?)
  - Stack overflow (what could cause a stack overflow? Ask the programmers.)
  - Unusual timing condition? (Can we create a script that lets us adjust timing of our input to the fridge?)
  - Unusual collection of things in the fridge?
- If you had a real customer who reported this problem, you MIGHT be able to get some of this information from them. But in risk-based testing, you don't have that customer. You just have to work backwards from a hypothetical failure to the conditions that might have produced it. Each set of conditions defines a new test.

How to map from a test idea to a test?

- When it is not clear how to work backwards to the relevant test, four tactics sometimes help:
  - Ask someone for help
  - Ask Google for help. (Look for discussions of the type of failure; look for discussions of different faults and see what types of failures they yield)
  - Review your toolkit of techniques, searching for a test type with relevant characteristics. (For example, if you think it might be a timing problem, what techniques help you focus on timing issues?)
  - Turn the failure into a story and gradually evolve the story into something you can test from. (This is what we did with Joe and the Fridge. A story is easier for some people to work with than a technologically equivalent, but inhuman, description of a failure.
- There are no guarantees in this, but you get better at it as you practice, and as you build a broader inventory of techniques.

SO, HOW DO WE DESIGN <u>RISK-BASED</u> TESTS?

Risk-based testing

QuickTests:
Simple,
Risk-Derived,
Test Techniques

QuickTests?

A ***quicktest*** is a cheap test that has some value but requires little preparation, knowledge, or time to perform.

- Participants at the 7th Los Altos Workshop on Software Testing (Exploratory Testing, 1999) pulled together a collection of these.
- James Whittaker published another collection in *How to Break Software.*
- Elisabeth Hendrickson teaches courses on bug hunting techniques and tools, many of which are quicktests or tools that support them.

A Classic QuickTest: The Shoe Test

Find an input field, move the cursor to it, put your shoe on the keyboard, and go to lunch.

Basically, you're using the auto-repeat on the keyboard for a cheap stress test.

- **Tests like this often overflow input buffers.**

In Bach's favorite variant, he finds a dialog box so constructed that pressing a key leads to, say, another dialog box (perhaps an error message) that also has a button connected to the same key that returns to the first dialog box.

- **This will expose some types of long-sequence errors (stack overflows, memory leaks, etc.)**

Another Classic Example of a QuickTest

Traditional boundary testing

- All you need is the variable, and its possible values.
- You need very little information about the meaning of the variable (why people assign values to it, what it interacts with).
- You test at boundaries because miscoding of boundaries is a common error.

Note the foundation-level assumption of this test:

Assumption

This is a programming error so common that it's worth building a test technique optimized to find errors of that type.

Why do we care about quicktests?

Point A: You imagine a way the program could fail.

Point B: You have to figure out how to design a test that could generate that failure.

Getting from Point A to Point B is a creative process. It depends on your ability to imagine a testing approach that could yield the test that yields the failure.

The more test techniques you know, and the better you understand them, the easier this creative task becomes.

- This is not some mysterious tester's intuition
- "Luck favors the mind that is prepared." (Louis Pasteur)

Quicktests give us straightforward, useful examples of tests that are focused on easy application of an underlying theory of error. They are just what we need to learn about, to start stretching our imagination.

“Attacks” to expose common coding errors

Jorgensen & Whittaker pulled together a collection of common coding errors, many of them involving insufficiently or incorrectly constrained variables.

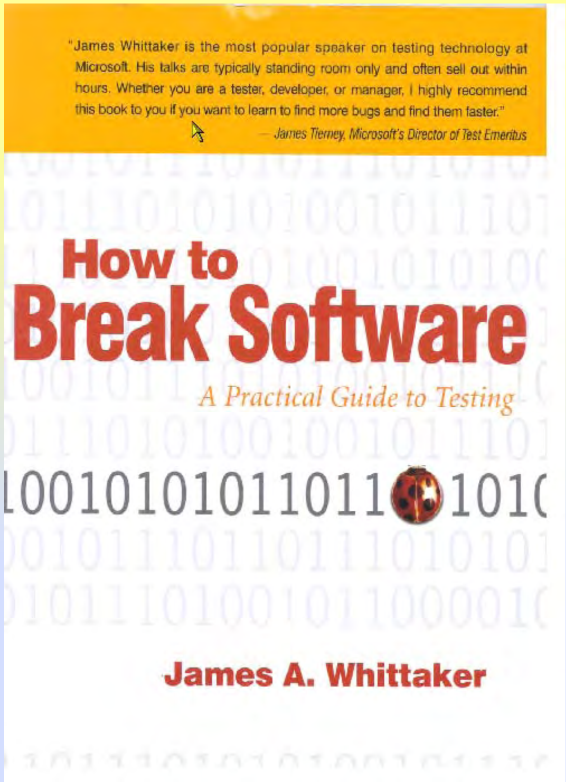
They created (or identified common) attacks to test for these.

An **attack** is a stereotyped class of tests, **optimized around a specific type of error.**

Think back to boundary testing:

- Boundary testing for numeric input fields is an example of an attack. The error is mis-specification (or mis-typing) of the upper or lower bound of the numeric input field.

"Attacks" to expose common coding errors



"James Whittaker is the most popular speaker on testing technology at Microsoft. His talks are typically standing room only and often sell out within hours. Whether you are a tester, developer, or manager, I highly recommend this book to you if you want to learn to find more bugs and find them faster."

— *James Tierney, Microsoft's Director of Test Emeritus*

In his book, How to Break Software, Professor Whittaker expanded the list and, for each attack, discussed

- When to apply it
- What software errors make the attack successful
- How to determine if the attack exposed a failure
- How to conduct the attack, and
- An example of the attack.

We'll list How to Break Software's attacks here, but recommend the book's full discussion.

“Attacks” to expose common coding errors

User interface attacks: Exploring the input domain

- Attack 1: Apply inputs that force all the error messages to occur
- Attack 2: Apply inputs that force the software to establish default values
- Attack 3: Explore allowable character sets and data types
- Attack 4: Overflow input buffers
- Attack 5: Find inputs that may interact and test combinations of their values
- Attack 6: Repeat the same input or series of inputs numerous times

» From Whittaker, How to Break Software

“Attacks” to expose common coding errors

User interface attacks: Exploring outputs

- Attack 7: Force different outputs to be generated for each input
- Attack 8: Force invalid outputs to be generated
- Attack 9: Force properties of an output to change
- Attack 10: Force the screen to refresh.

» From Whittaker, How to Break Software

“Attacks” to expose common coding errors

Testing from the user interface: Data and computation

Exploring stored data

- Attack 11: Apply inputs using a variety of initial conditions
- Attack 12: Force a data structure to store too many or too few values
- Attack 13: Investigate alternate ways to modify internal data constraints

» From Whittaker, How to Break Software

“Attacks” to expose common coding errors

Testing from the user interface: Data and computation

Exploring computation and feature interaction

- Attack 14: Experiment with invalid operand and operator combinations
- Attack 15: Force a function to call itself recursively
- Attack 16: Force computation results to be too large or too small
- Attack 17: Find features that share data or interact poorly

» From Whittaker, How to Break Software

“Attacks” to expose common coding errors

System interface attacks

Testing from the file system interface: Media-based attacks

- Attack 1: Fill the file system to its capacity
- Attack 2: Force the media to be busy or unavailable
- Attack 3: Damage the media

Testing from the file system interface: File-based attacks

- Attack 4: Assign an invalid file name
- Attack 5: Vary file access permissions
- Attack 6: Vary or corrupt file contents

» From Whittaker, How to Break Software

Additional QuickTests from LAWST

Several of the tests we listed at LAWST (7th Los Altos Workshop on Software Testing, 1999) are equivalent to the attacks later published by Whittaker.

He develops the attacks well, and we recommend his descriptions.

In addition, LAWST generated several other quicktests, including some that aren't directly tied to a simple fault model.

Many of the ideas in these notes were reviewed and extended by the other LAWST 7 participants: Brian Lawrence, III, Jack Falk, Drew Pritsker, Jim Bampos, Bob Johnson, Doug Hoffman, Chris Agruss, Dave Gelperin, Melora Svoboda, Jeff Payne, James Tierney, Hung Nguyen, Harry Robinson, Elisabeth Hendrickson, Noel Nyman, Bret Pettichord, & Rodney Wilson. We appreciate their contributions.

Additional QuickTests: Interference testing

We look at asynchronous events here. One task is underway, and we do something to interfere with it.

In many cases, the critical event is extremely time sensitive. For example:

- An event reaches a process just as, just before, or just after it is timing out or just as (before / during / after) another process that communicates with it will time out listening to this process for a response. ("Just as?"—if special code is executed in order to accomplish the handling of the timeout, "just as" means during execution of that code)
- An event reaches a process just as, just before, or just after it is servicing some other event.
- An event reaches a process just as, just before, or just after a resource needed to accomplish servicing the event becomes available or unavailable.

Additional QuickTests: Interference testing

Generate interrupts

- From a device related to the task
  - e.g. pull out a paper tray, perhaps one that isn’t in use while the printer is printing
- From a device unrelated to the task
  - e.g. move the mouse and click while the printer is printing
- From a software event
  - e.g. set another program's (or this program's) time-reminder to go off during the task under test

Additional QuickTests Interference testing

Change something this task depends on

- swap out a floppy
- change the contents of a file that this program is reading
- change the printer that the program will print to (without signaling a new driver)
- change the video resolution

Additional QuickTests: Interference testing

Cancel

- Cancel the task
  - at different points during its completion
- Cancel some other task while this task is running
  - a task that is in communication with this task (the core task being studied)
  - a task that will eventually have to complete as a prerequisite to completion of this task
  - a task that is totally unrelated to this task

Additional QuickTests: Interference testing

Pause: Find some way to create a temporary interruption in the task.

Pause the task

- for a short time
- for a long time (long enough for a timeout, if one will arise)

For example,

- Put the printer on local
- Put a database under use by a competing program, lock a record so that it can't be accessed — yet.

Additional QuickTests: Interference testing

Swap (out of memory)

- Swap the process out of memory while it's running
  - (e.g. change focus to another application; keep loading or adding applications until the application under test is paged to disk.)
  - Leave it swapped out for 10 minutes or whatever the timeout period is. Does it come back? What is its state? What is the state of processes that are supposed to interact with it?
  - Leave it swapped out much longer than the timeout period. Can you get it to the point where it is supposed to time out, then send a message that is supposed to be received by the swapped-out process, then time out on the time allocated for the message? What are the resulting state of this process and the one(s) that tried to communicate with it?
- Swap a related process out of memory while the process under test is running.

Additional QuickTests: Interference testing

Compete

- Compete for a device (such as a printer)
  - put device in use, then try to use it from software under test
  - start using device, then use it from other software
  - If there is a priority system for device access, use software that has higher, same and lower priority access to the device before and during attempted use by software under test
- Compete for processor attention
  - some other process generates an interrupt (e.g. ring into the modem, or a time-alarm in your contact manager)
  - try to do something during heavy disk access by another process
- Send this process another job while one is underway

Additional QuickTests: Follow up recent changes

Code changes cause side effects

- Test the modified feature / change itself.
- Test features that interact with this one.
- Test data that are related to this feature or data set.
- Test scenarios that use this feature in complex ways.

Even More QuickTests

Quick tours of the program

- **Variability Tour**: Tour a product looking for anything that is variable and vary it. Vary it as far as possible, in every dimension possible.

  Exploring variations is part of the basic structure of Bach's testing when he first encounters a product.

- **Complexity Tour**: Tour a product looking for the most complex features and data. Create complex files.
- **Sample Data Tour**: Employ any sample data you can, and all that you can. The more complex the better.

  - (from Bach & Bolton's Rapid Testing Course)

Even More QuickTests (from Bach / Bolton)

- **Continuous Use**: While testing, do not reset the system. Leave windows and files open. Let disk and memory usage mount. You're hoping the system ties itself in knots over time.
- **Adjustments:** Set some parameter to a certain value, then, at any later time, reset that value to something else without resetting or recreating the containing document or data structure.
- **Dog Piling:** Get more processes going at once; more states existing concurrently. Nested dialog boxes and non-modal dialogs provide opportunities to do this.
- **Undermining**: Start using a function when the system is in an appropriate state, then change the state part way through (for instance, delete a file while it is being edited, eject a disk, pull net cables or power cords) to an inappropriate state. This is similar to interruption, except you are expecting the function to interrupt itself by detecting that it no longer can proceed safely.

Even More QuickTests (from Bach / Bolton)

- **Error Message Hangover**: Make error messages happen. Test hard after they are dismissed. Developers often handle errors poorly. Bach once broke into a public kiosk by right clicking rapidly after an error occurred. It turned out the security code left a 1/5 second window of opportunity for me to access a special menu and take over the system.
- **Click Frenzy**: Testing is more than "banging on the keyboard", but that phrase wasn't coined for nothing. Try banging on the keyboard. Try clicking everywhere. Bach broke into a touchscreen system once by poking every square centimeter of every screen until he found a secret button.
- **Multiple Instances**: Run a lot of instances of the application at the same time. Open the same files.

Even More QuickTests (from Bach / Bolton)

- **Feature Interactions**: Discover where individual functions interact or share data. Look for any interdependencies. Tour them. Stress them. Bach once crashed an app by loading up all the fields in a form to their maximums and then traversing to the report generator.
- **Cheap Tools!** Learn how to use InCtrl5, Filemon, Regmon, AppVerifier, Perfmon, Task Manager (all of which are free). Have these tools on a thumb drive and carry it around. Also, carry a digital camera. Bach carries a tiny 3 megapixel camera and a tiny video camera in his coat pockets. He uses them to record screen shots and product behaviors.
  - Elisabeth Hendrickson suggests several additional tools at http://www.bughunting.com/bugtools.html
- **Resource Starvation**: Progressively lower memory and other resources until the product gracefully degrades or ungracefully collapses.

Even More QuickTests (from Bach / Bolton)

- **Play "Writer Sez":** Look in the online help or user manual and find instructions about how to perform some interesting activity. Do those actions. Then improvise from them. Often writers are hurried as they write down steps, or the software changes after they write the manual.
- **Crazy Configs**: Modify O/S configuration in non-standard or non-default ways either before or after installing the product. Turn on "high contrast" accessibility mode, or change the localization defaults. Change the letter of the system hard drive.
- **Grokking**: Find some aspect of the product that produces huge amounts of data or does some operation very quickly. For instance, look a long log file or browse database records very quickly. Let the data go by too quickly to see in detail, but notice trends in length or look or shape of the patterns as you see them.

Parlour tricks are not risk-free

These tricks can generate lots of flash in a hurry

- The DOS disk I/O example
- The Amiga clicky-click-click-click example

As political weapons, they are double-edged

- If people realize what you're doing, you lose credibility
- Anyone you humiliate becomes a potential enemy

Some people (incorrectly) characterize exploratory testing as if it were a collection of quicktests.

As test design tools, they are like good candy

- Yummy
- Everyone likes them
- Not very nutritious. (You never get to the deeper issues of the program.)

SO, HOW DO WE DESIGN RISK-BASED TESTS?

Risk: The possibility of suffering harm or loss

In software testing, we think of risk on three dimensions:

- A way the program could fail (technically, this is the **hazard,** or the **failure mode,** but I'll often refer to this as the risk because that is so common among testers)
- How likely it is that the program could fail in that way
- What the consequences of that failure could be

For testing purposes, the most important is:

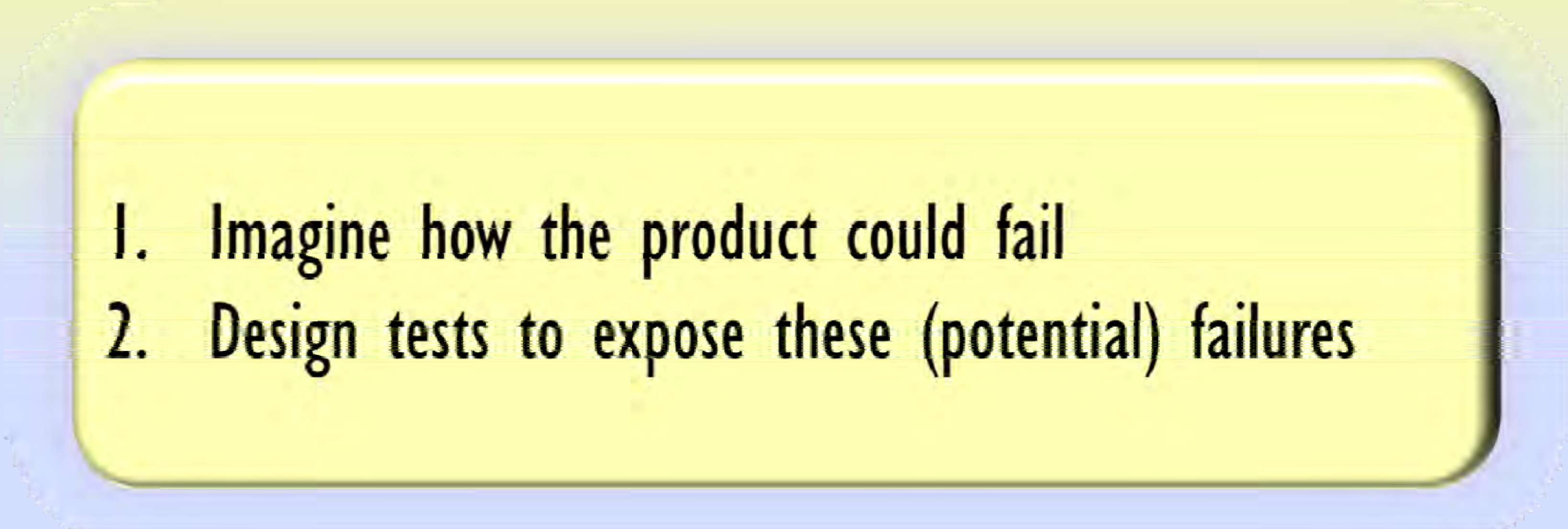
- *A way the program could fail*

For project management purposes,

- *How likely*
- *What consequences*

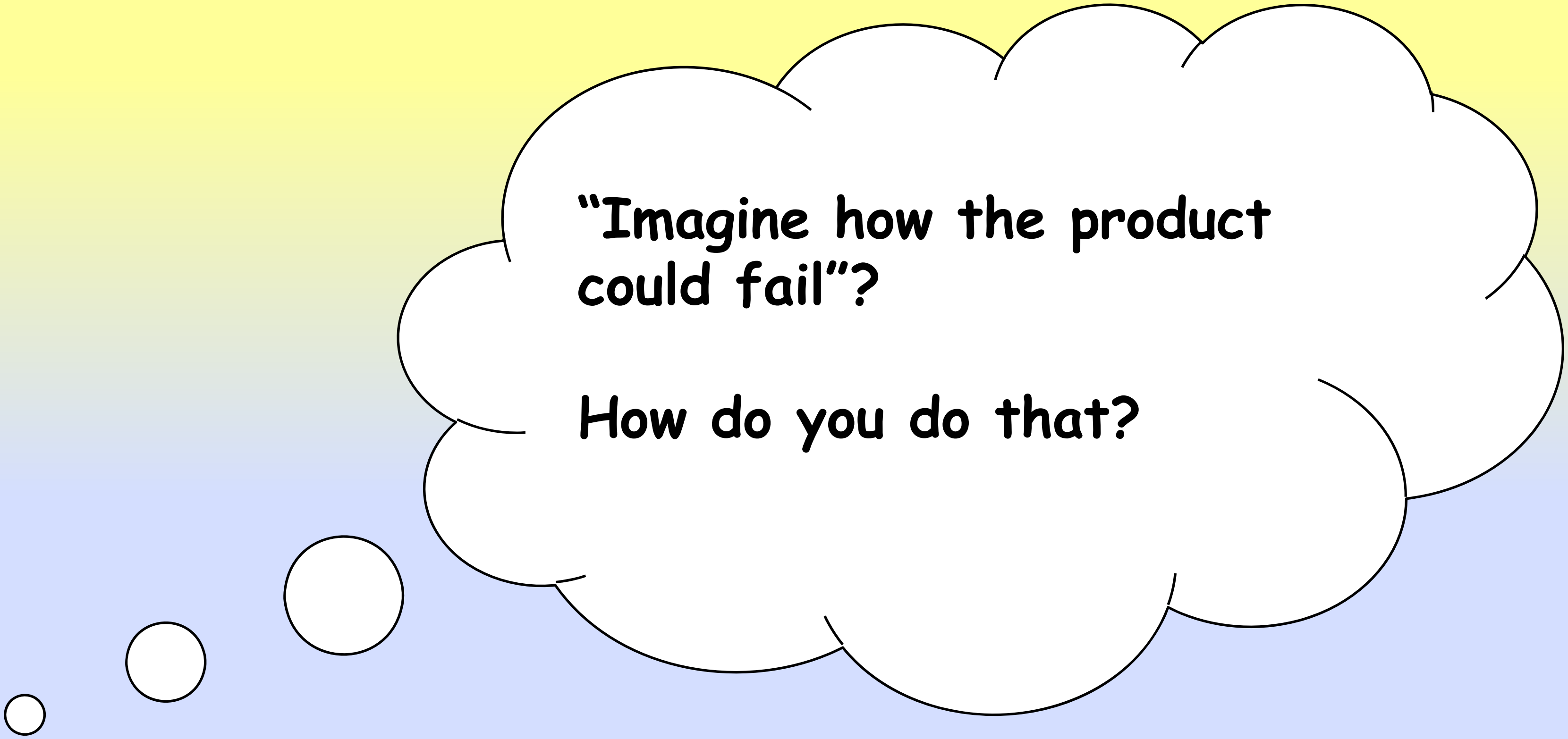
For testing: A way the program could fail

The essence of risk-based **testing** is this:



1. Imagine how the product could fail
2. Design tests to expose these (potential) failures

Just one little problem



"Imagine how the product could fail"?

How do you do that?

Just one problem

"Imagine how the product could fail" ?
How do you do that?

We'll consider three classes of heuristics:

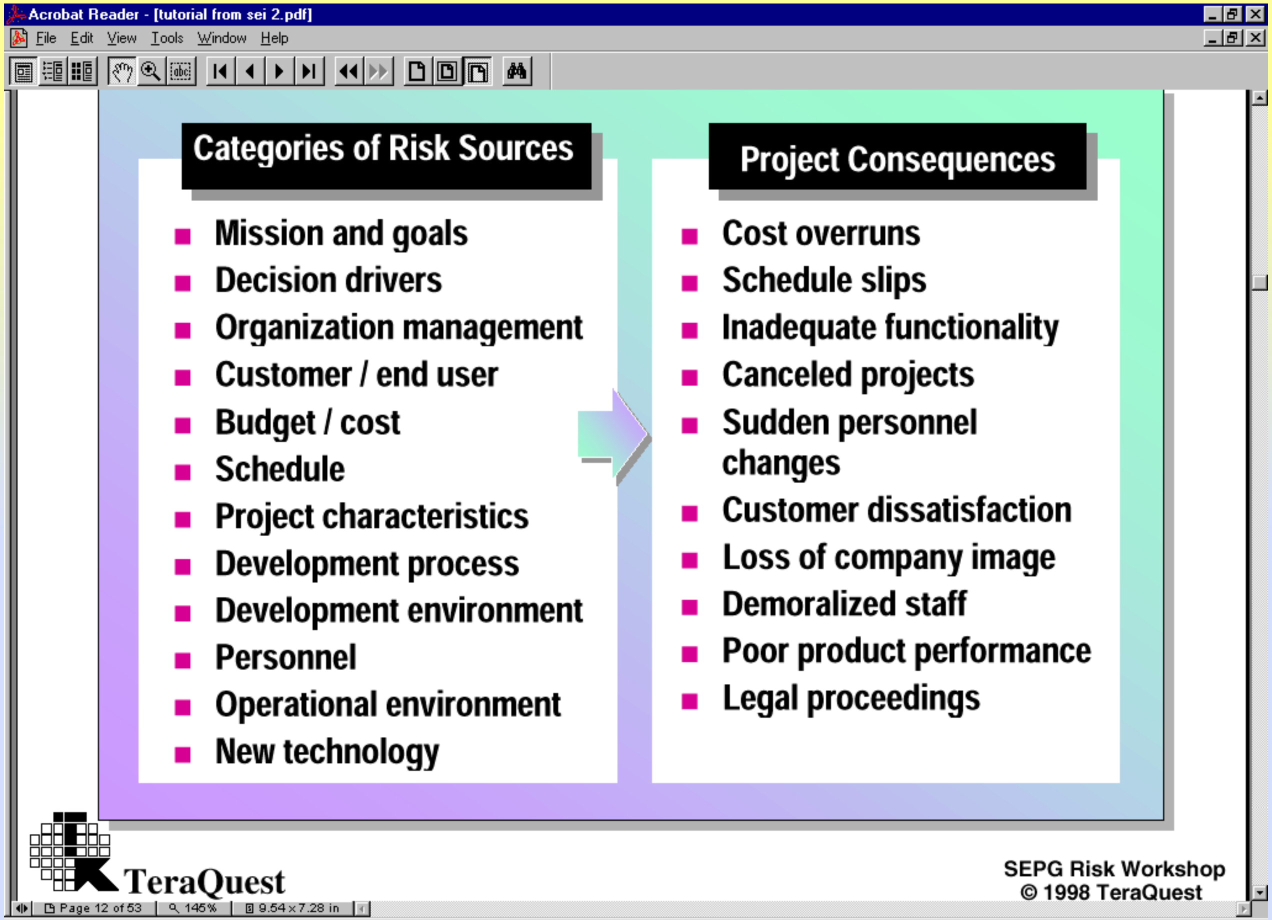
- Apply common techniques (quicktests or attacks) to take advantage of common errors (we did that already)
- Recognize common project warning signs (and test things associated with the risky aspects of the project).
- Apply failure mode and effects analysis to (many or all) elements of the product and to the product's key quality criteria.

We call these heuristics because they are fallible but useful guides. You have to exercise your own judgment about which to use when.

Risk-based testing

Project-Level Risk Factors

Classic, project-level risk analysis


Acrobat Reader - [tutorial from sei 2.pdf]
File Edit View Tools Window Help
Page 12 of 53
145%
9.54 x 7.28 in

Categories of Risk Sources

- Mission and goals
- Decision drivers
- Organization management
- Customer / end user
- Budget / cost
- Schedule
- Project characteristics
- Development process
- Development environment
- Personnel
- Operational environment
- New technology

Project Consequences

- Cost overruns
- Schedule slips
- Inadequate functionality
- Canceled projects
- Sudden personnel changes
- Customer dissatisfaction
- Loss of company image
- Demoralized staff
- Poor product performance
- Legal proceedings

TeraQuest

SEPG Risk Workshop
© 1998 TeraQuest

Project-level risk analyses usually consider risk factors that can make the project as a whole fail, and how to manage those risks.

Project-level risk analysis

Project risk management involves

- Identification of the different risks to the project (issues that might cause the project to fail or to fall behind schedule or to cost too much or to dissatisfy customers or other stakeholders)
- Analysis of the potential costs associated with each risk
- Development of plans and actions to reduce the likelihood of the risk or the magnitude of the harm
- Continuous assessment or monitoring of the risks (or the actions taken to manage them)

Useful material available free at http://seir.sei.cmu.edu

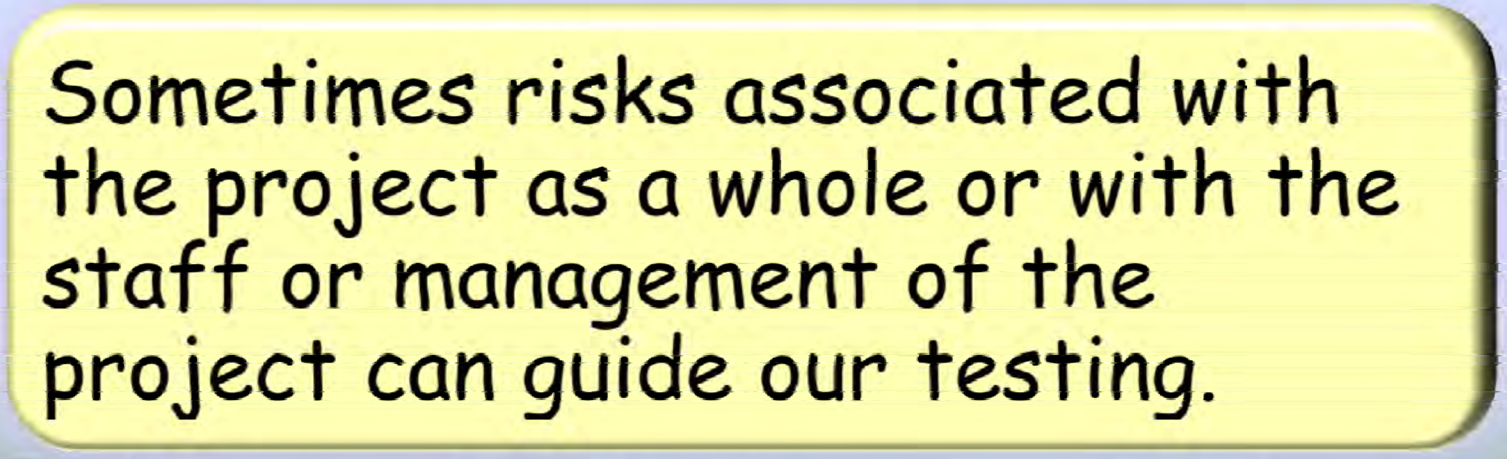
http://www.coyotevalley.com (Brian Lawrence)

The problem for our purposes is that this level of analysis doesn't give us much guidance as to how to test.

Project-level risk analysis

- Might not give us much guidance about **how** to test
- But it might give us a lot of hints about **where** to test

- If you can imagine a potential failure
- In many cases, that failure might be possible at many different places in the program
- Which should you try first?



Sometimes risks associated with the project as a whole or with the staff or management of the project can guide our testing.

Project risk heuristics: Where to look for errors

New things: less likely to have revealed its bugs yet.

New technology: same as new code, plus the risks of unanticipated problems.

Learning curve: people make more mistakes while learning.

Changed things: same as new things, but changes can also break old code.

Poor control: without SCM, files can be overridden or lost.

Late change: rushed decisions, rushed or demoralized staff lead to mistakes.

Rushed work: some tasks or projects are chronically underfunded and all aspects of work quality suffer.

Fatigue: tired people make mistakes.

Distributed team: a far flung team communicates less

Project risk heuristics: Where to look for errors

Other staff issues: alcoholic, mother died, two programmers who won’t talk to each other (neither will their code)…

Surprise features: features not carefully planned may have unanticipated effects on other features.

Third-party code: external components may be much less well understood than local code, and much harder to get fixed.

Unbudgeted: unbudgeted tasks may be done shoddily.

Ambiguous: ambiguous descriptions (in specs or other docs) lead to incorrect or conflicting implementations.

Conflicting requirements: ambiguity often hides conflict, result is loss of value for some person.

Project risk heuristics: Where to look for errors

Mysterious silence: when something interesting or important is not described or documented, it may have not been thought through, or the designer may be hiding its problems.

Unknown requirements: requirements surface throughout development. Failure to meet a legitimate requirement is a failure of quality for that stakeholder.

Evolving requirements: people realize what they want as the product develops. Adhering to a start-of-the-project requirements list may meet the contract but yield a failed product.

Buggy: anything known to have lots of problems has more.

Recent failure: anything with a recent history of problems.

Upstream dependency: may cause problems in the rest of the system

Downstream dependency: sensitive to problems in the rest of the system.

Project risk heuristics: Where to look for errors

Distributed: anything spread out in time or space, that must work as a unit.

Open-ended: any function or data that appears unlimited.

Complex: what's hard to understand is hard to get right.

Language-typical errors: such as wild pointers in C.

Little system testing: untested software will fail.

Little unit testing: programmers normally find and fix most of their own bugs.

Previous reliance on narrow testing strategies: can yield a many-version backlog of errors not exposed by those techniques.

Weak test tools: if tools don't exist to help identify / isolate a class of error (e.g. wild pointers), the error is more likely to survive to testing and beyond.

Project risk heuristics: Where to look for errors

Unfixable: bugs that survived because, when they were first reported, no one knew how to fix them in the time available.

Untestable: anything that requires slow, difficult or inefficient testing is probably undertested.

Publicity: anywhere failure will lead to bad publicity.

Liability: anywhere that failure would justify a lawsuit.

Critical: anything whose failure could cause substantial damage.

Precise: anything that must meet its requirements exactly.

Project risk heuristics: Where to look for errors

Easy to misuse: anything that requires special care or training to use properly.

Popular: anything that will be used a lot, or by a lot of people.

Strategic: anything that has special importance to your business.

VIP: anything used by particularly important people.

Visible: anywhere failure will be obvious and upset users.

Invisible: anywhere failure will be hidden and remain undetected until a serious failure results.

Project risk heuristics: Where to look for errors

If you have access to the source code, and have programming skills, take a look at work on prediction of failure-prone files and modules by:

- Dolores and Wayne Zage (Ball State University, SERC)
- Emmet James Whitehead (UC Santa Cruz), for example

S. Kim, E. J. Whitehead, Jr., and Y. Zhang, "Classifying Software Changes: Clean or Buggy?," *IEEE Transactions on Software Engineering, to appear, 2008, manuscript* available at http://www.cs.ucsc.edu/~ejw/papers/cc.pdf.

This is very recent, and I think very promising, empirical research.

Risk-based testing

Failure Modes

Failure Mode & Effects Analysis (FMEA)

Failure mode: A way that the program could fail

Example: Portion of analysis for an installer product

- Wrong files installed
  - temporary files not cleaned up
  - old files not cleaned up after upgrade
  - unneeded file installed
  - needed file not installed
  - correct file installed in the wrong place
- Files clobbered
  - older file replaces newer file
  - user data file clobbered during upgrade
- Other apps clobbered
  - file shared with another product is modified
  - file belonging to another product is deleted

Failure mode & effects analysis

Widely used for safety analysis of goods.

Consider the product in terms of its components. For each component

- Imagine the ways it could fail. For each potential failure (each failure mode), ask questions:
  - What would that failure look like?
  - How would you detect that failure?
  - How expensive would it be to search for that failure?
  - Who would be impacted by that failure?
  - How much variation would there be in the effect of the failure?
  - How serious (on average) would that failure be?
  - How expensive would it be to fix the underlying cause?
- On the basis of the analysis, decide whether it is cost effective to search for this potential failure

Failure mode & effects analysis (FMEA)

Several excellent web pages introduce FMEA and SFMEA (software FMEA)

- http://www.fmeainfocentre.com/
- http://www.fmeainfocentre.com/presentations/**SFMEA**-IIE.pdf
- http://www.fmeainfocentre.com/papers/mackel1.pdf
- http://www.quality-one.com/services/fmea.php
- http://www.visitask.com/fmea.asp
- http://healthcare.isixsigma.com/library/content/c040317a.asp
- http://www.qualitytrainingportal.com/resources/fmea/
- http://citeseer.ist.psu.edu/69117.html

FMEA

As some of the papers / presentations on the preceding slide note, one of the key weaknesses of FMEA in practice is:

- It is hard to identify all the ways the product can fail
- So we get a long, but not necessarily thorough list of failure modes
- This can be misleading

- In traditional industries (e.g. automotive), this type of analysis is guided by long experience with failures in the field and failures discovered in manufacturing
- In the absence of strong records for a particular product, how do we generate a strong failure mode list for software?
- The next several subsections of this presentation, leading up to Bach's heuristic test strategy model, provide a series of ideas

Bug catalogs

Testing Computer Software included an appendix that listed almost 500 common bugs (actually, failure modes).

The list evolved across several products and companies. It was intended to be a generic list, more of a starting point for failure mode planning than a complete list.

To be included in the list:

- A particular failure mode had to be possible in at least two significantly different programs
- A particular failure mode had to be possible in applications running under different operating systems (we occasionally relaxed this rule)

You can find the TCS 2nd edition list (appendix) on Hung Nguyen's site: http://www.logigear.com/resources/articles_lg/Common_Software_Errors.pdf?fileid=2458

Bug catalogs

Testing Computer Software included an appendix that listed almost 500 common bugs (actually, failure modes).

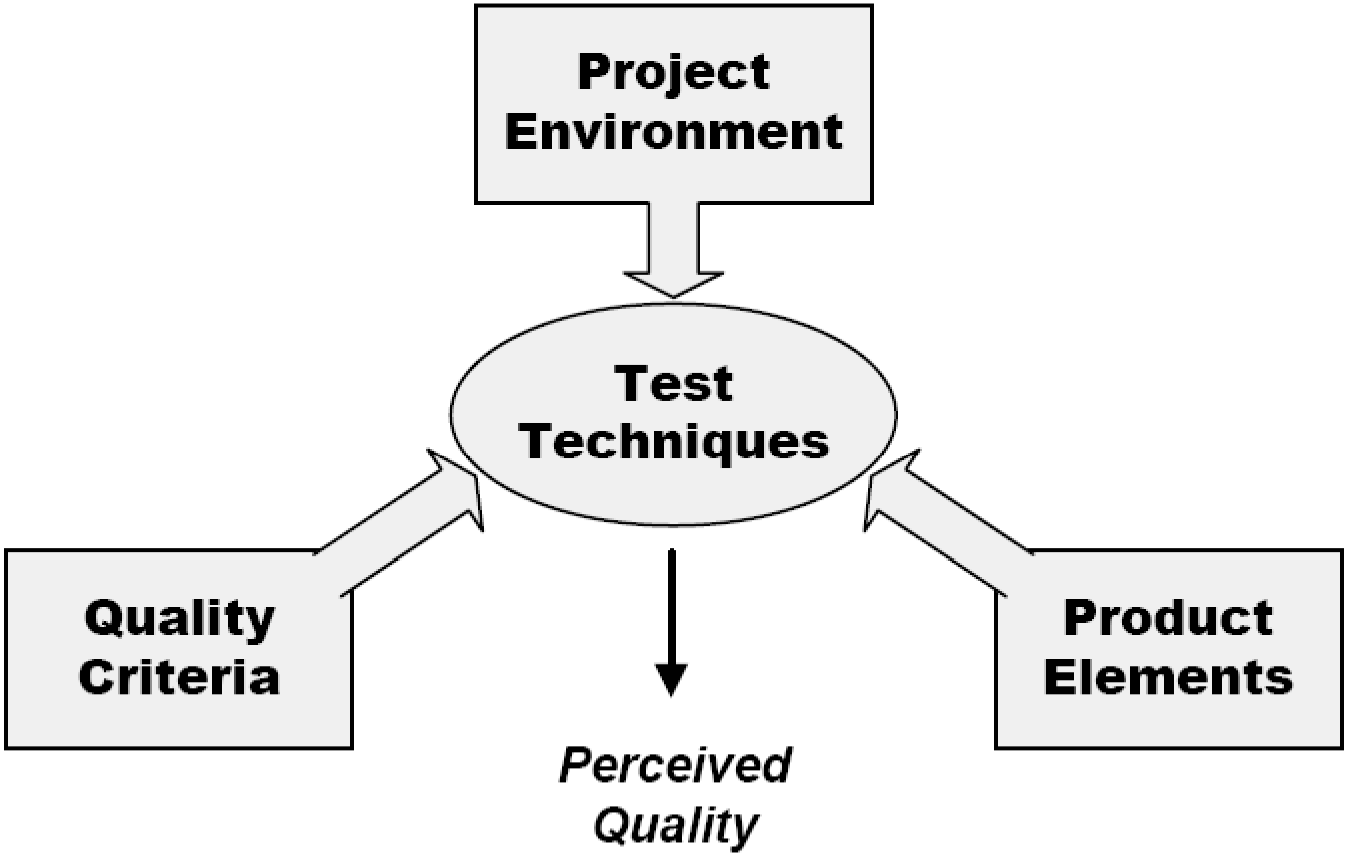
Some people found this appendix very useful for training staff, generating test ideas and supporting auditing of test plans,

However, it was

- organized idiosyncratically,
- its coverage was uneven, and
- some people inappropriately treated it as a comprehensive list (because they didn’t understand it, or were unable to do the independent critical analysis needed to tailor this to their application)

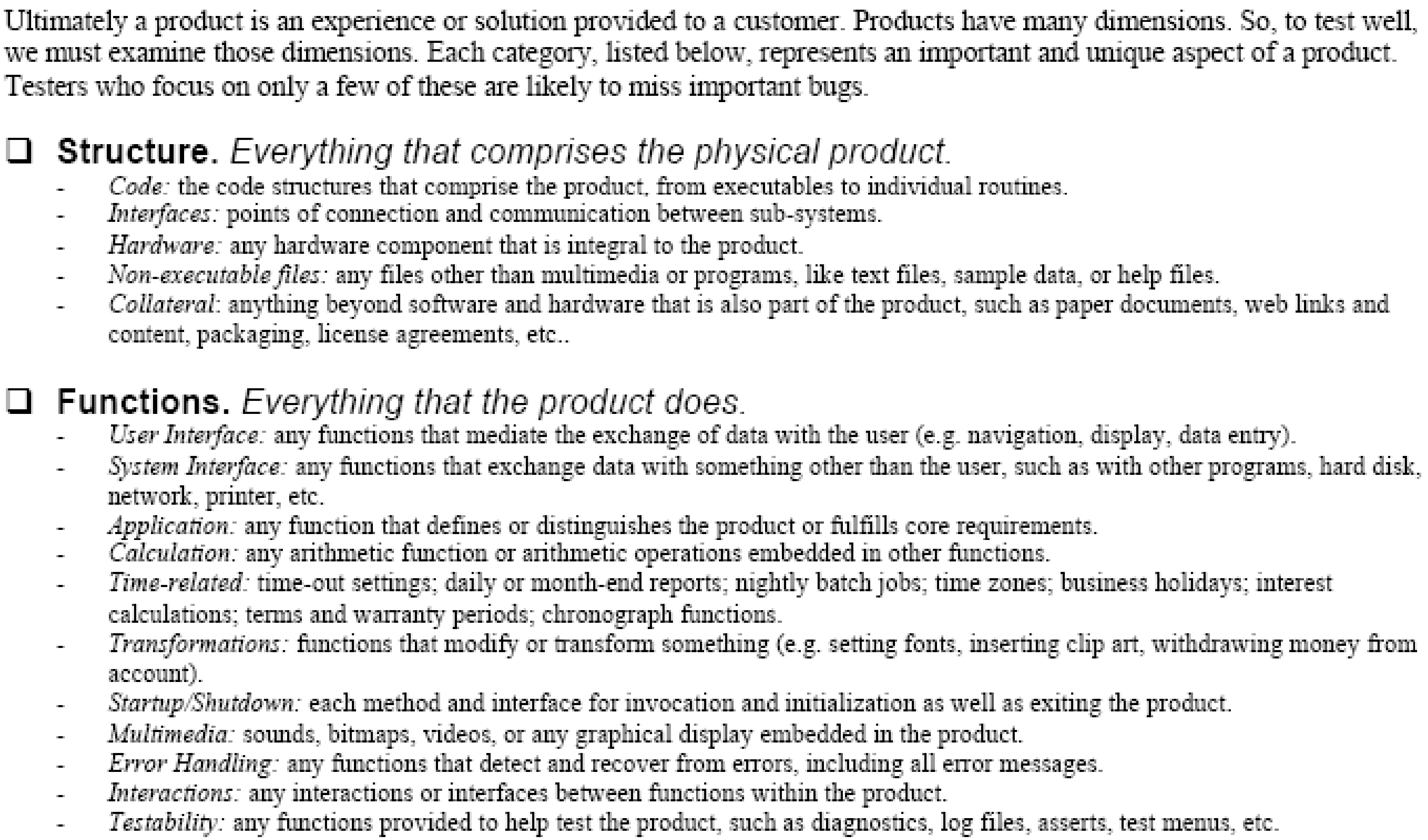
Eventually, I stopped recommending this list (even though I developed the first edition of it and had found it very useful for several years) in favor of an early version of James Bach’s Heuristic test strategy model (latest version at http://www.satisfice.com/tools/satisfice-tsm-4p.pdf)

Heuristic Test Strategy Model



http://www.satisfice.com/tools/satisfice-tsm-4p.pdf

Product Elements



Ultimately a product is an experience or solution provided to a customer. Products have many dimensions. So, to test well, we must examine those dimensions. Each category, listed below, represents an important and unique aspect of a product. Testers who focus on only a few of these are likely to miss important bugs.

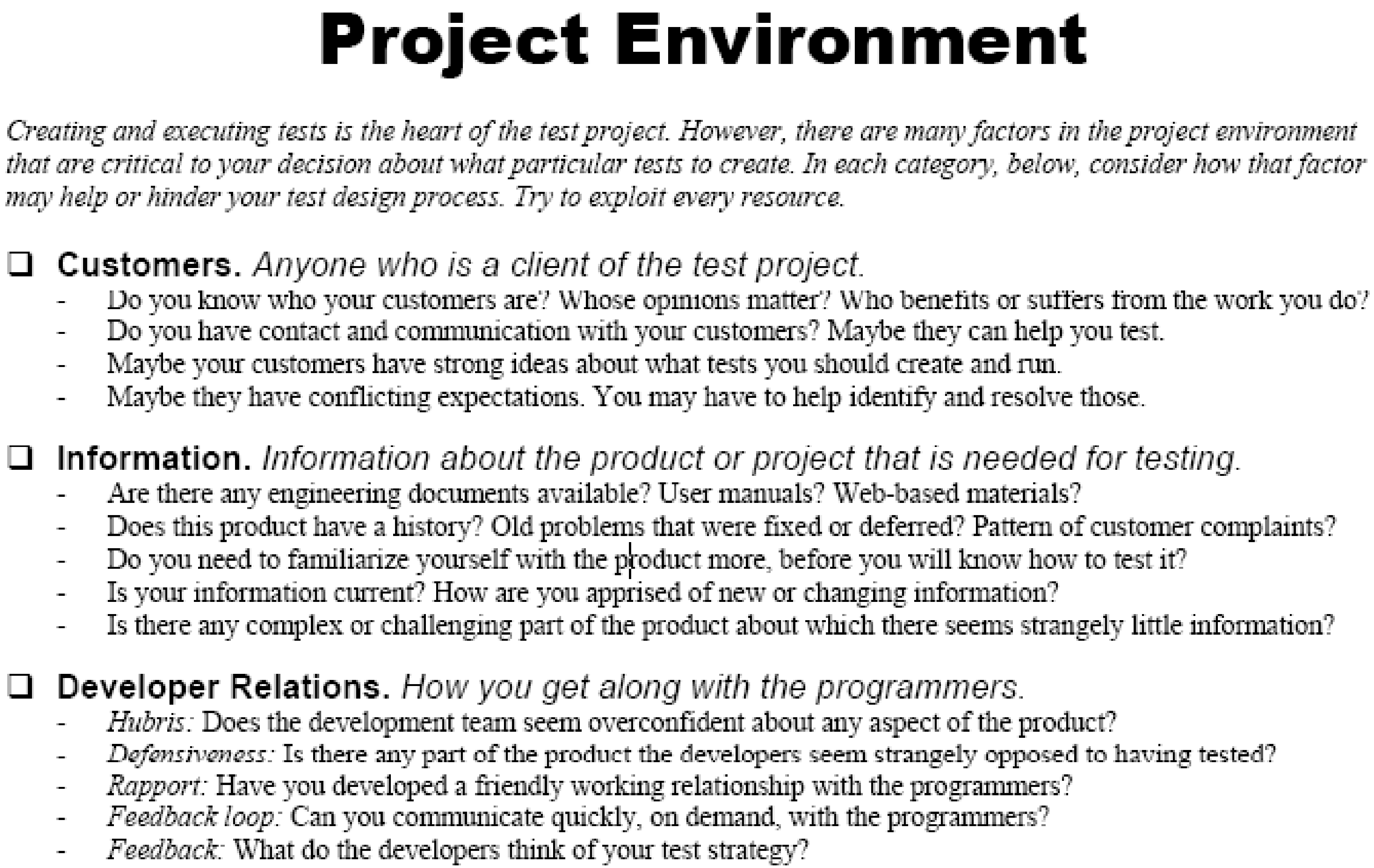
❑ **Structure.** *Everything that comprises the physical product.*

- *Code:* the code structures that comprise the product, from executables to individual routines.
- *Interfaces:* points of connection and communication between sub-systems.
- *Hardware:* any hardware component that is integral to the product.
- *Non-executable files:* any files other than multimedia or programs, like text files, sample data, or help files.
- *Collateral:* anything beyond software and hardware that is also part of the product, such as paper documents, web links and content, packaging, license agreements, etc..

❑ **Functions.** *Everything that the product does.*

- *User Interface:* any functions that mediate the exchange of data with the user (e.g. navigation, display, data entry).
- *System Interface:* any functions that exchange data with something other than the user, such as with other programs, hard disk, network, printer, etc.
- *Application:* any function that defines or distinguishes the product or fulfills core requirements.
- *Calculation:* any arithmetic function or arithmetic operations embedded in other functions.
- *Time-related:* time-out settings; daily or month-end reports; nightly batch jobs; time zones; business holidays; interest calculations; terms and warranty periods; chronograph functions.
- *Transformations:* functions that modify or transform something (e.g. setting fonts, inserting clip art, withdrawing money from account).
- *Startup/Shutdown:* each method and interface for invocation and initialization as well as exiting the product.
- *Multimedia:* sounds, bitmaps, videos, or any graphical display embedded in the product.
- *Error Handling:* any functions that detect and recover from errors, including all error messages.
- *Interactions:* any interactions or interfaces between functions within the product.
- *Testability:* any functions provided to help test the product, such as diagnostics, log files, asserts, test menus, etc.

Structure / Functions / Data / Platform / Operations / Time



Project Environment

Creating and executing tests is the heart of the test project. However, there are many factors in the project environment that are critical to your decision about what particular tests to create. In each category, below, consider how that factor may help or hinder your test design process. Try to exploit every resource.

□ **Customers.** *Anyone who is a client of the test project.*

- Do you know who your customers are? Whose opinions matter? Who benefits or suffers from the work you do?
- Do you have contact and communication with your customers? Maybe they can help you test.
- Maybe your customers have strong ideas about what tests you should create and run.
- Maybe they have conflicting expectations. You may have to help identify and resolve those.

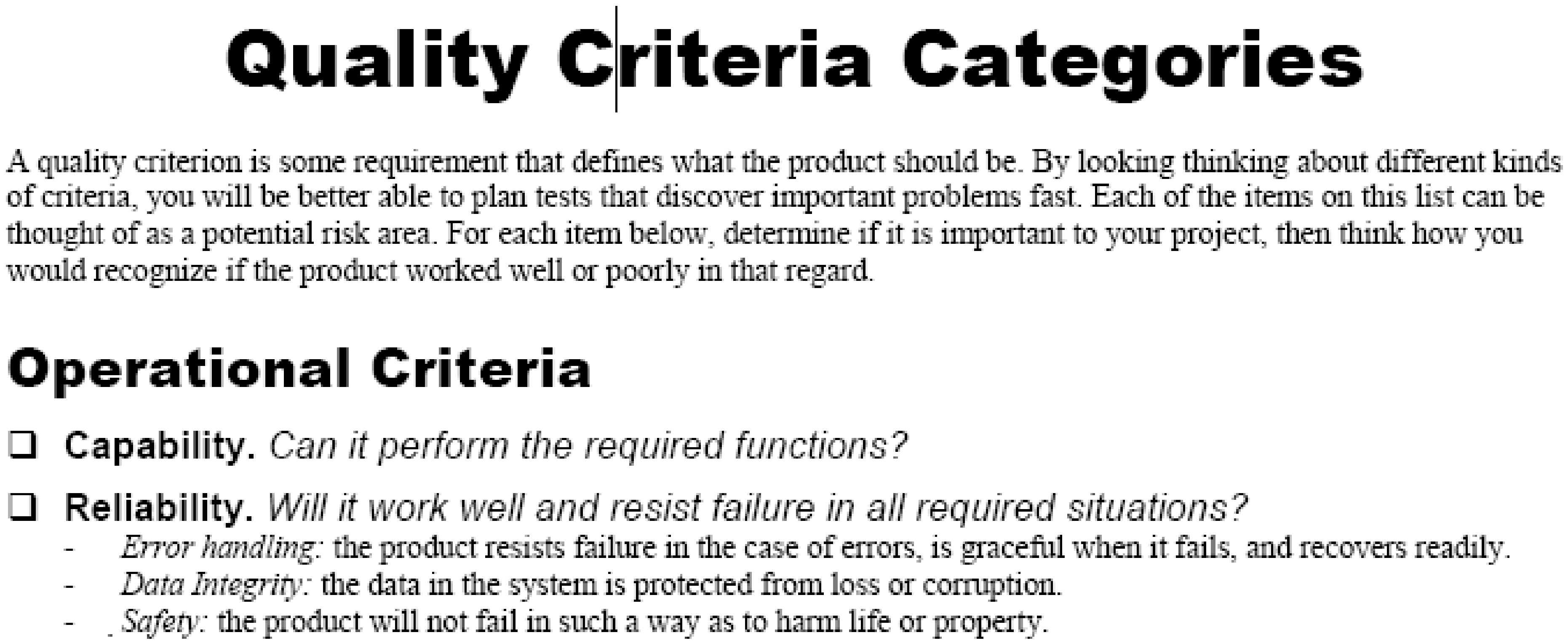
□ **Information.** *Information about the product or project that is needed for testing.*

- Are there any engineering documents available? User manuals? Web-based materials?
- Does this product have a history? Old problems that were fixed or deferred? Pattern of customer complaints?
- Do you need to familiarize yourself with the product more, before you will know how to test it?
- Is your information current? How are you apprised of new or changing information?
- Is there any complex or challenging part of the product about which there seems strangely little information?

□ **Developer Relations.** *How you get along with the programmers.*

- *Hubris:* Does the development team seem overconfident about any aspect of the product?
- *Defensiveness:* Is there any part of the product the developers seem strangely opposed to having tested?
- *Rapport:* Have you developed a friendly working relationship with the programmers?
- *Feedback loop:* Can you communicate quickly, on demand, with the programmers?
- *Feedback:* What do the developers think of your test strategy?

Customers / Information / Developer relations / Test team / Equipment & tools / Schedule / Test items / Deliverables



Quality Criteria Categories

A quality criterion is some requirement that defines what the product should be. By looking thinking about different kinds of criteria, you will be better able to plan tests that discover important problems fast. Each of the items on this list can be thought of as a potential risk area. For each item below, determine if it is important to your project, then think how you would recognize if the product worked well or poorly in that regard.

Operational Criteria

❑ **Capability.** *Can it perform the required functions?*

❑ **Reliability.** *Will it work well and resist failure in all required situations?*

- *Error handling:* the product resists failure in the case of errors, is graceful when it fails, and recovers readily.
- *Data Integrity:* the data in the system is protected from loss or corruption.
- *Safety:* the product will not fail in such a way as to harm life or property.

<u>Operational criteria</u>: Capability / Reliability / Usability / Security / Scalability / Performance / Installability / Compatibility

<u>Development criteria</u>: Supportability / Testability / Maintainability / Portability / Localizability

Notes on the Heuristic Test Strategy Model

- The individual elements (“Customers”, “Capability”, etc.) are Guide Words.
- A lot of work has been done applying different types of guide words in safety critical applications (HAZOPS depends fundamentally on guidewords.
- See United States Coast Guard. *Risk-based Decision-making Guidelines*. [accessed 2008 March 3]; Available from: http://www.uscg.mil/hq/g-m/risk/e-guidelines/hazop.htm for discussion of HAZOPS and other safety-critical test/analysis techniques
- The model is extensive but not exhaustive. Giri and Ajay (see next slide) both had to customize for their applications. We (including Bach) all expected this.

Building a failure mode catalog

Giri Vijayaraghavan and Ajay Jha followed similar approaches in developing failure mode catalogs for their M.Sc. theses (available in the lab publications set at www.testingeducation.org):

- Identify components
  - They used the Heuristic Test Strategy Model as a starting point.
  - Imagine ways the program could fail (in this component).
    - They used magazines, web discussions, some corporations' bug databases, interviews with people who had tested their class of products, and so on, to guide their imagination.
  - Imagine failures involving interactions among components
- They did the same thing for quality attributes (see next section).

These catalogs are not orthogonal. They help generate test ideas, but are not suited for classifying test ideas.

Building failure mode lists from product elements: Shopping cart example

Think in terms of the components of your product

- **Structures: Everything that comprises the logical or physical product**
  - Database server
  - Cache server
- **Functions: Everything the product does**
  - Calculation
  - Navigation
  - Memory management
  - Error handling
- **Data: Everything the product processes**
  - Human error (retailer)
  - Human error (customer)
- **Operations: How the product will be used**
  - Upgrade
  - Order processing
- **Platforms: Everything on which the product depends**

How could they fail?

» Adapted from Giri Vijayaraghavan's Master's thesis.

FMEA & quality attributes

In FMEA, we list a bunch of things (components of the product under test) we could test, and then figure out how they might fail.

Quality attributes cut across the components:

- **Usability**
  - Easy to learn
  - Reasonable number of steps
  - Accessible to someone with a disability
    - Auditory
    - Visual
      - *Imagine evaluating every product element in terms of accessibility to someone with a visual impairment.*

Using a failure mode list

Test idea generation

- Find a potential bug (failure mode) in the list
- Ask whether the software under test could have this bug
- If it is theoretically possible that the program could have the bug, ask how you could find the bug if it was there.
- Ask how plausible it is that this bug could be in the program and how serious the failure would be if it was there.
- If appropriate, design a test or series of tests for bugs of this type.

Using a failure mode list

Test plan auditing

- Pick categories to sample from
- From each category, find a few potential defects in the list
- For each potential defect, ask whether the software under test could have this defect
- If it is theoretically possible that the program could have the defect, ask whether the test plan could find the bug if it was there.

Getting unstuck

- Look for classes of problem outside of your usual box

Training new staff

- Expose them to what can go wrong, challenge them to design tests that could trigger those failures

Risk-based testing: Some papers of interest

- Stale Amland, Risk Based Testing, http://www.amland.no/WordDocuments/EuroSTAR99Paper.doc
- James Bach, Reframing Requirements Analysis
- James Bach, Risk and Requirements- Based Testing
- James Bach, James Bach on Risk-Based Testing
- Stale Amland & Hans Schaefer, Risk based testing, a response (at http://www.satisfice.com)
- Stale Amland's course notes on Risk-Based Agile Testing (December 2002) at http://www.testingeducation.org/coursenotes/amland_stale/cm_200212_exploratorytesting
- Carl Popper, Conjectures & Refutations
- James Whittaker, How to Break Software
- Giri Vijayaraghavan's papers and thesis on bug taxonomies, at http://www.testingeducation.org/articles

About Cem Kaner

- Professor of Software Engineering, Florida Tech
- Research Fellow at Satisfice, Inc.

I've worked in all areas of product development (programmer, tester, writer, teacher, user interface designer, software salesperson, organization development consultant, as a manager of user documentation, software testing, and software development, and as an attorney focusing on the law of software quality.)

Senior author of three books:

- *Lessons Learned in Software Testing* (with James Bach & Bret Pettichord)
- *Bad Software* (with David Pels)
- *Testing Computer Software* (with Jack Falk & Hung Quoc Nguyen).

My doctoral research on psychophysics (perceptual measurement) nurtured my interests in human factors (usable computer systems) and measurement theory.