

# Risk-Based Testing: Some Basic Concepts

QAI Managers Workshop, April 2008

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

These notes are partially based on research that was supported by NSF Grant CCLI-0717613  
"Adaptation & Implementation of an Activity-Based Online or Hybrid Course in Software Testing."  
Any opinions, findings and conclusions or recommendations expressed in this material are those of  
the author(s) and do not necessarily reflect the views of the National Science Foundation.

## For more:

My website, [www.testingeducation.org/BBST](http://www.testingeducation.org/BBST) provides a large set of videos and slides on testing concepts and techniques. These materials are FREE: you can download them and use them in your own courses or in study groups with your coworkers.

The CD includes some of these materials along with a few additional source documents:

- Videos and slides (and papers) on
  - Risk-based testing, including M.Sc. Theses by Ajay Jha and Giri Vijayaraghavan illustrating failure mode analyses for software.
  - Specification-based testing
  - Scripted testing
- Slides and papers on
  - Exploratory testing

# Risk: The possibility of suffering harm or loss

In software testing, we often 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*

## Risk-based testing?

**Risk-based test prioritization:** Evaluate each area of a product and allocate time/money according to perceived risk.

**Risk-based test lobbying:** Use information about risk to justify requests for more time / staff.

**Risk-based test design:** 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. The most powerful tests are the ones that maximize a program's opportunity to fail.

## *Risk-based prioritization:*

Evaluate each area of a product and allocate time/money according to perceived risk.

# Risk-based prioritization

Often called “risk-based testing” or “risk-based test management”

If one part of the program is “higher risk” than another, allocate more time/money for testing of it.

So how do we estimate risk?

# Silly mathemagics

For each area (e.g. feature):

- Estimate the probability it will fail, on a scale of 1 (very low) to 5 (very high)
- Estimate the severity of the failure if it will happen, on a scale of 1 (lowest) to 5 (highest)
- Estimate risk as rated-probability x rated-severity.
- Allocate more resource and earlier testing to bigger-numbered areas.

# Silly mathemagics

Estimate risk as rated-probability x rated-severity?

1. It is mathematically meaningless to multiply orderings (1 to 5) because the distance from 1 to 2 is incomparable with the distance from (for example) 2 to 3.
2. The probability that an area will fail is 100%. Everything has a bug or two (or more). Eventually, if you test/use long enough, you will find it.
3. The severity is richly multidimensional, for example:
  - How many people are affected?
  - What is the cost per failure?
  - How embarrassing is the failure?
  - How long will it take to fix this bug if it is there? (If you don't find long-to-fix bugs early, they don't get fixed)



# Risk-based prioritization: The goal is...

*Prioritization....*

- What should you do first?
- How much should you budget?
- When should you stop?

*.... that makes sense to other people*

My experience:

- The high-risk and low-risk areas are relatively easy to identify and justify
- The mid-level ones are harder, but the ranking of them often comes most naturally from other factors (how long it would take to fix code in this part of the product; who is available when to fix these bugs; how squeaky the local wheels are wrt this feature, etc.)

# Classic, project-level risk analysis

The screenshot shows a presentation slide within an Acrobat Reader window. The slide is divided into two main sections: 'Categories of Risk Sources' on the left and 'Project Consequences' on the right, connected by a large blue arrow pointing from left to right. The 'Categories of Risk Sources' list includes: Mission and goals, Decision drivers, Organization management, Customer / end user, Budget / cost, Schedule, Project characteristics, Development process, Development environment, Personnel, Operational environment, and New technology. The 'Project Consequences' list includes: Cost overruns, Schedule slips, Inadequate functionality, Canceled projects, Sudden personnel changes, Customer dissatisfaction, Loss of company image, Demoralized staff, Poor product performance, and Legal proceedings. The slide footer contains the TeraQuest logo and the text 'SEPG Risk Workshop © 1998 TeraQuest'. The Acrobat Reader window title is 'Acrobat Reader - [tutorial from sei 2.pdf]' and the status bar shows 'Page 12 of 53', '145%' zoom, and '9.54x7.28 in' dimensions.

**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)

# 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:

- 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 lobbying:*

Use information about risk to justify requests  
for more time / staff.

# Risk-based lobbying

Quality is value to some person (Weinberg)

Different people have different valuations of different parts of the product.

If you want more people / time / money to test some part of the product:

- think about who will suffer most if that part of the product doesn't work well
- help them understand the kinds of tests you COULD run of their favored area
- help them understand some of the reasons you have of being mistrustful of this implementation of this area
- scenarios (scenario tests) are often helpful for illustration
- Let the person who will be most impacted by the bug champion your need for resources

## *Risk-based test-design:*

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. The most powerful tests are the ones that maximize a program's opportunity to fail

# For risk-based test design and execution:

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

“I imagine how the product could fail” ?  
How do you do that?

We'll consider three classes of heuristics:

- Recognize common project warning signs (and test things associated with the risky aspects of the project).
- Apply common techniques (quicktests or attacks) to take advantage of common errors
- 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

*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.

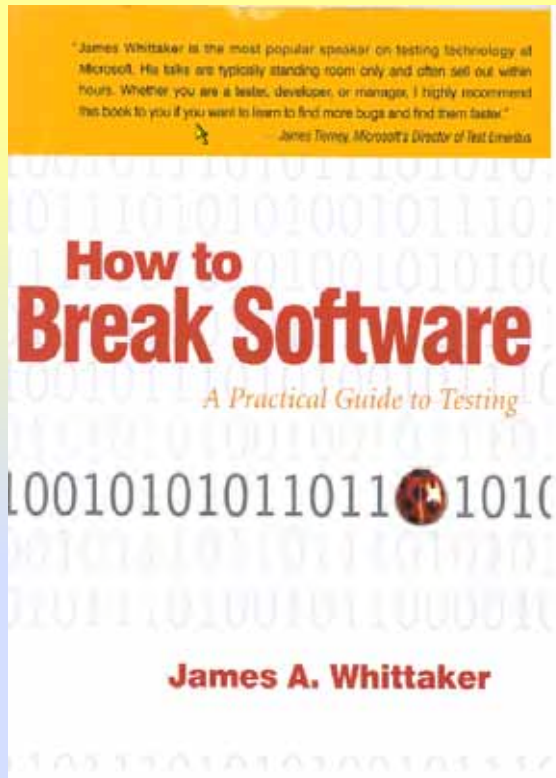
They created (or identified common) attacks to test for these.

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



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

# 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-II.E.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>

# 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 2<sup>nd</sup> edition list (appendix) on Hung Nguyen's site:  
[http://www.logigear.com/resources/articles\\_Ig/Common\\_Software\\_Errors.pdf?fileid=2458](http://www.logigear.com/resources/articles_Ig/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> )

# 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](http://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.

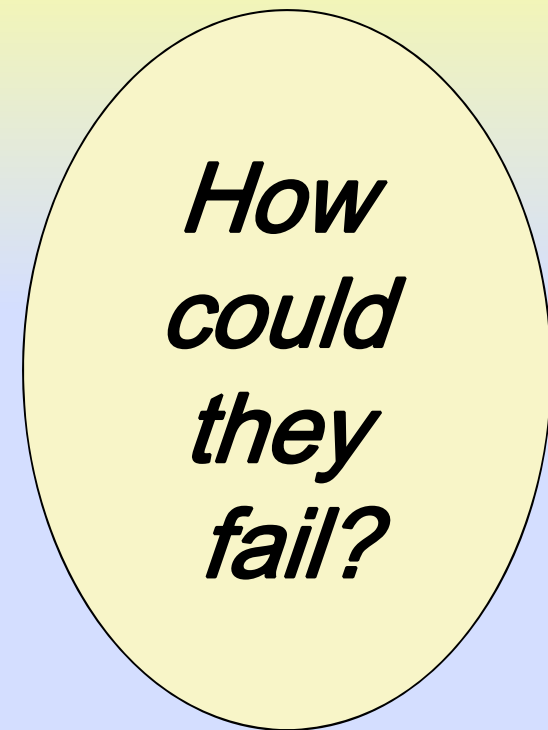


## Building failure mode lists from product elements: Shopping cart example

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**

» 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**

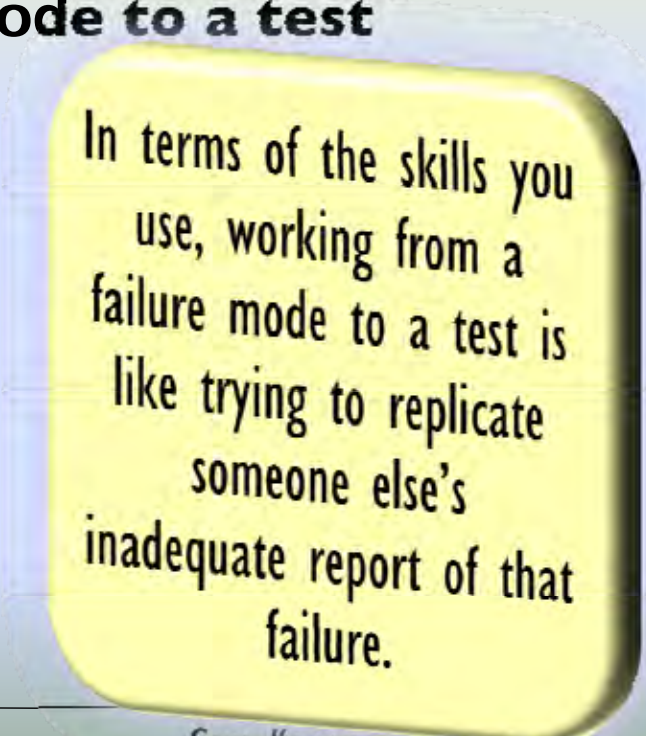
- Expose them to what can go wrong, challenge them to design tests that could trigger those failures

# 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](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>

# 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?



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.

# 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.