Software Testing as a Social Science

Cem Kaner, J.D., Ph.D.

Presentation at

STEP 2008

Memphis, May 2008

What are we actually looking for?

- Programmers find and fix most of their own bugs
- What testers find are what programmers missed.
- Testers are looking for the bugs that hide in programmers' blind spots.
- To test effectively, our theories of error have to be theories about the mistakes people make and when / why they make them.

Social Science?

Social sciences study humans, especially humans in society.

- What will the impact of X be on people?
- Work with qualitative & quantitative research methods.
- High tolerance for ambiguity, partial answers, situationally specific results.
- Ethics / values issues are relevant.
- Diversity of values / interpretations is normal.
- Observer bias is an accepted fact of life and is managed explicitly in well-designed research.

Testing-related work involves:

- Discovery
- Communication
- Persuasion
- Control
- Measurement
- Accounting
- Project management

We use tools to do these, but is testing about the tools or about the objectives?

Are the tools the objective?

Are the processes the objective?

Let's look at another example: Test-related metrics

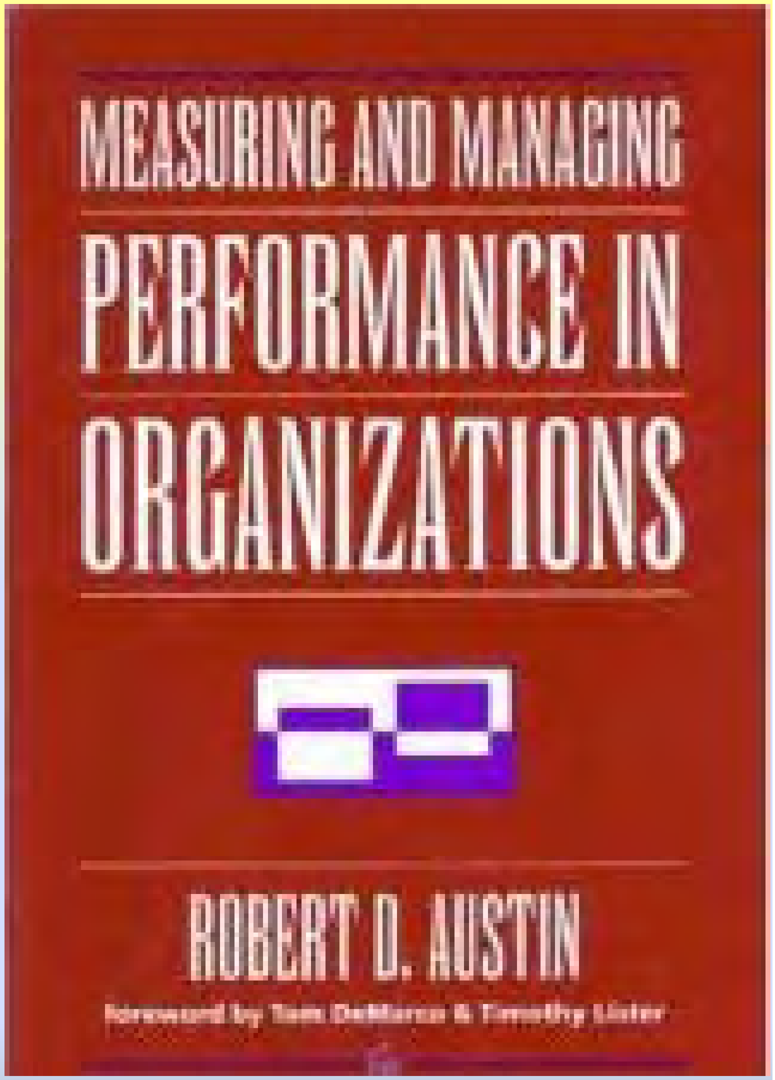
Test-related metrics

Most testing metrics are human performance metrics

- How productive is this tester?
- How good is her work?
- How good is someone else's work?
- How long is this work taking them?

These are well studied questions in the social sciences and not well studied when we ignore the humans and fixate on the computer.

We ignore the human issues at risk.



MEASURING AND MANAGING PERFORMANCE IN ORGANIZATIONS

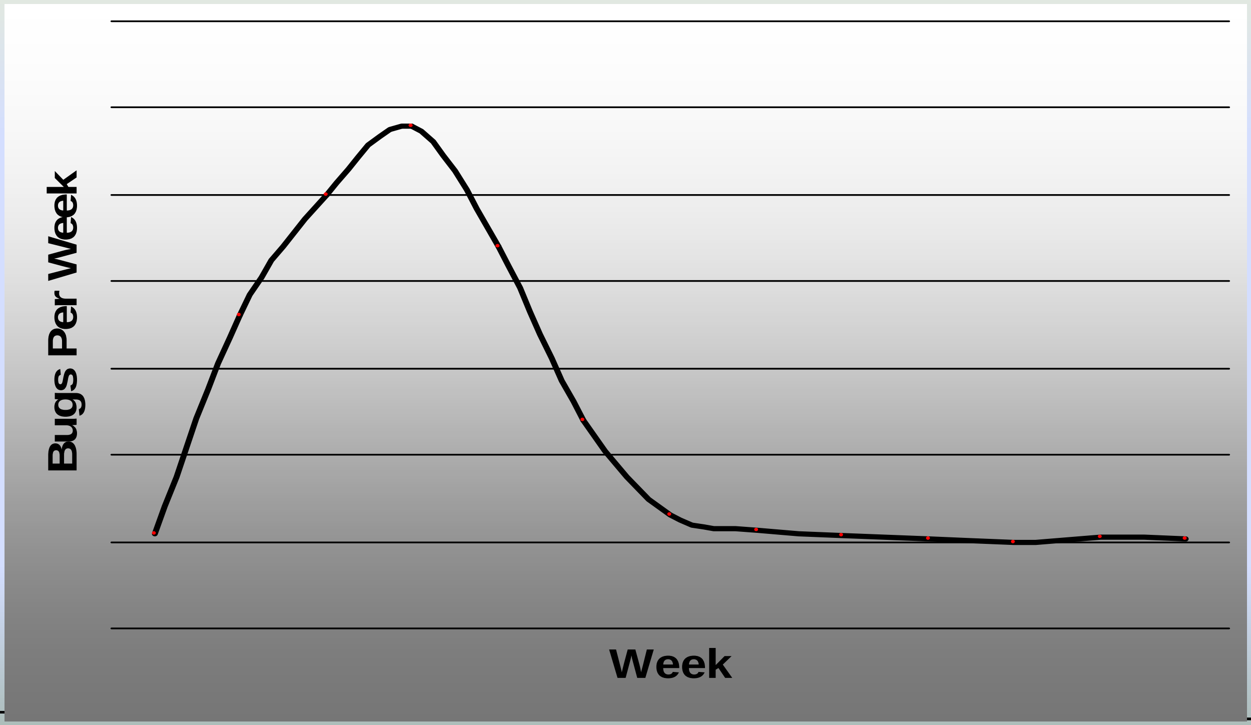
ROBERT D. AUSTIN

Foreword by Tom DeMarco & Timothy Lister

Example: Bug find rates

Some people measure completeness of testing with bug curves:

- New bugs found per week ("Defect arrival rate")
- Bugs still open (each week)
- Ratio of bugs found to bugs fixed (per week)



Weibull reliability model

Bug curves can be useful progress indicators, but some people fit the data to theoretical curves to determine when the project will complete.

The model's assumptions

1. Testing occurs in a way similar to the way the software will be operated.
2. All defects are equally likely to be encountered.
3. Defects are corrected instantaneously, without introducing additional defects.
4. All defects are independent.
5. There is a fixed, finite number of defects in the software at the start of testing.
6. The time to arrival of a defect follows the Weibull distribution.
7. The number of defects detected in a testing interval is independent of the number detected in other testing intervals for any finite collection of intervals.

- See Erik Simmons, When Will We Be Done Testing? Software Defect Arrival Modeling with the Weibull Distribution.

The Weibull model

I think it's absurd to rely on a distributional model (or any model) when every assumption it makes about testing is obviously false.

One of the advocates of this approach points out that

"Luckily, the Weibull is robust to most violations."

- This illustrates the use of surrogate measures—we don't have an attribute description or model for the attribute we really want to measure, so we use something else, that is allegedly "robust", in its place. This can be very dangerous
- The Weibull distribution has a shape parameter that allows it to take a very wide range of shapes. If you have a curve that generally rises then falls (one mode), you can approximate it with a Weibull.

But how should we interpret an adequate fit to an otherwise indefensible model?

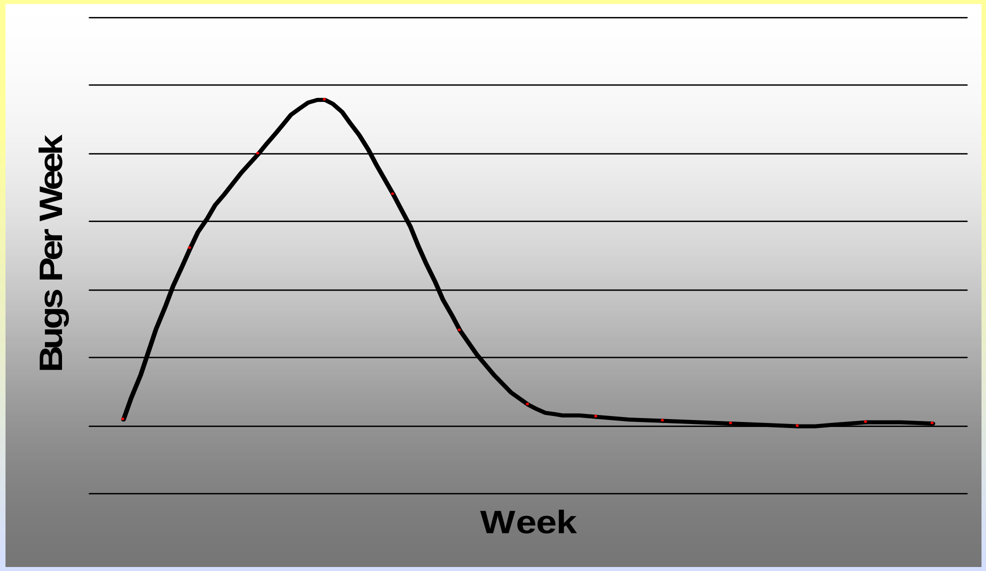
Side effects of bug curves

When development teams are pushed to show project bug curves that look like the Weibull curve, they are pressured to show a rapid rise in their bug counts, an early peak, and a steady decline of bugs found per week.

In practice, project teams, including testers, in this situation often adopt dysfunctional methods, doing things that will be bad for the project over the long run in order to make the numbers go up quickly.

- For more on measurement dysfunction, read Bob Austin's book, Measurement and Management of Performance in Organizations.
- For more observations of problems like these in reputable software companies, see Doug Hoffman's article, The Dark Side of Software Metrics.

Side effects of bug curves: Early testing



Predictions from these curves are based on parameters estimated from the data. You can start estimating the parameters once the curve has hit its peak and gone down a bit.

The sooner the project hits its peak, the earlier we would predict the product will ship.

So, early in testing, the pressure on testers is to drive the bug count up quickly, as soon as possible.

Side effects of bug curves

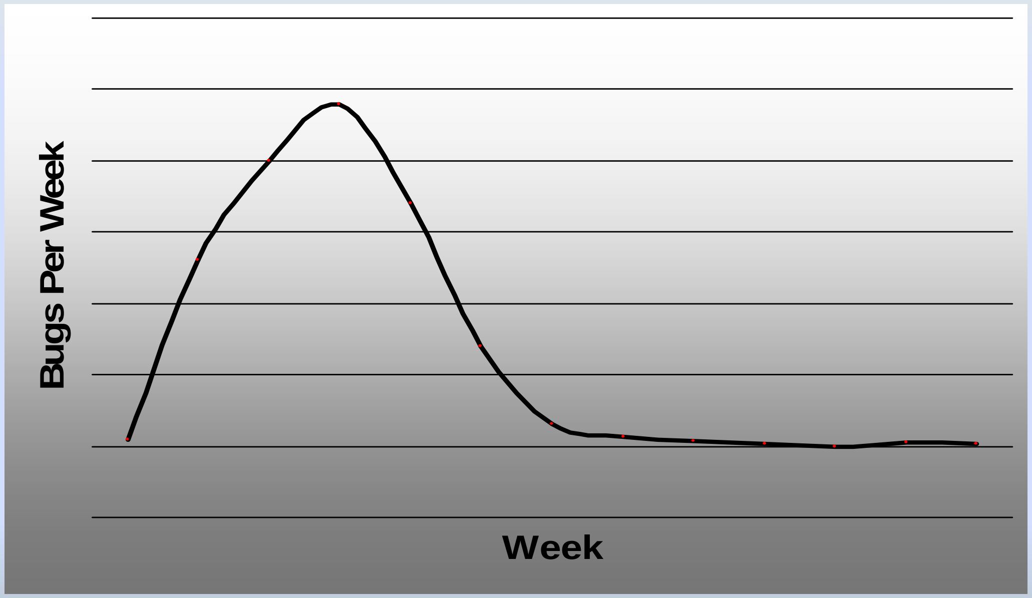
Earlier in testing, the pressure is to increase bug counts. In response, testers will:

- Run tests of features known to be broken or incomplete.
- Run multiple related tests to find multiple related bugs.
- Look for easy bugs in high quantities rather than hard bugs.
- Less emphasis on infrastructure, automation architecture, tools and more emphasis of bug finding. (Short term payoff but long term inefficiency.)

Side effects of bug curves: Later in testing

After we get past the peak, the expectation is that testers will find fewer bugs each week than they found the week before.

Based on the number of bugs found at the peak, and the number of weeks it took to reach the peak, the model can predict the later curve, how many bugs per week in each subsequent week.

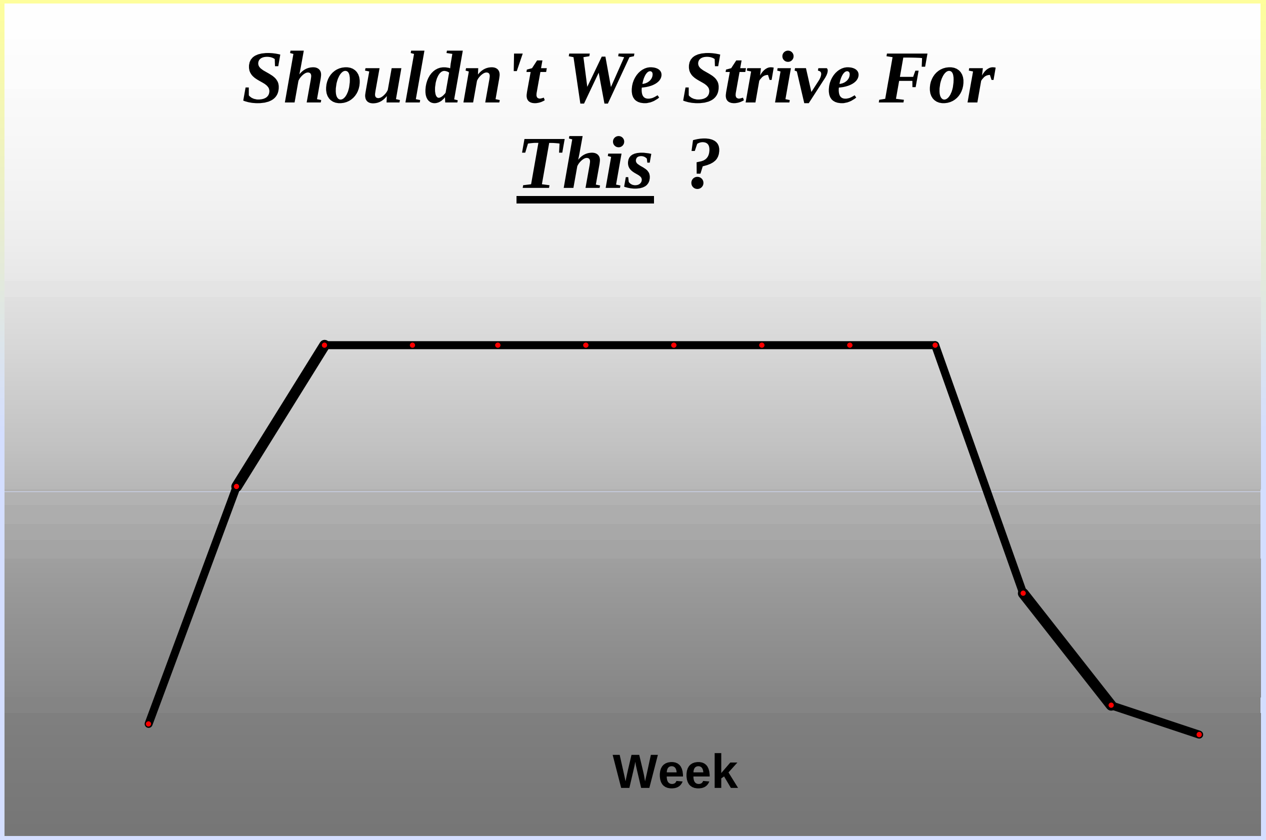


Side effects of bug curves

Later in testing, the pressure is to decrease the new bug rate:

- Run lots of already-run regression tests.
- Don't look as hard for new bugs.
- Shift focus to appraisal, status reporting.
- Classify unrelated bugs as duplicates.
- Class related bugs as duplicates (and closed), hiding key data about the symptoms / causes of the problem.
- Postpone bug reporting until after the measurement checkpoint (milestone). (Some bugs are lost.)
- Report bugs informally, keeping them out of the tracking system.
- Testers get sent to the movies before measurement checkpoints.
- Programmers ignore bugs they find until testers report them.
- Bugs are taken personally.
- More bugs are rejected.

Bad models are counterproductive



Testers live and breathe tradeoffs

The time needed for test-related tasks is infinitely larger than the time available.

For example, time you spend on

- - analyzing, troubleshooting, and effectively describing a failure

Is time no longer available for

- - Designing tests
- - Executing tests
- - Reviews, inspections
- - Retooling

- Documenting tests
- Automating tests
- Supporting tech support
- Training other staff

Mechanistic thinking about testing glosses over this fundamental complexity

Mechanistic Thinking? -- What's a Computer Program?

The last couple of years, I taught intro programming.

Texts define a "computer program" like this:

A program is a set of instructions for a computer

Computer Program

A set of instructions for a computer?

What about what the program is for?

Computer Program

A set of instructions for a computer?

What about what the program is for?

We could define a house

- as a set of construction materials
- assembled according to house-design patterns.

Computer Program

A set of instructions for a computer?

What about what the program is *for?*

We could define a house

- as a set of construction materials
- assembled according to house-design patterns.

But I'd rather define it as something built for people to live in.

Something built for people to live in…

The focus is on

- Intent
- Stakeholders

Set of instructions for a computer…

Where are the

- Intent?
- Stakeholders?

Ignore these and lay groundwork for the classic problems of software engineering …

in the first week of the first programming class.

A different definition

A computer program is

- a communication
- among several humans and computers
- who are distributed over space and time,
- that contains instructions that can be executed by a computer.

The point of the program is to provide value to the stakeholders.

Stakeholder

A person

- who is affected by
- the success or failure of a project
- or the actions or inactions of a product
- or the effects of a service.

Stakeholder

To know how to test something, you must understand

- who the stakeholders are and
- how they can be affected by the product or system under test.

Quality and errors

Quality is value to some person

-- Jerry Weinberg

Under this view:

- Quality is inherently subjective
  - Different stakeholders will perceive the same product as having different levels of quality

Testers look for different things …

… for different stakeholders

Software error

An attribute of a software product

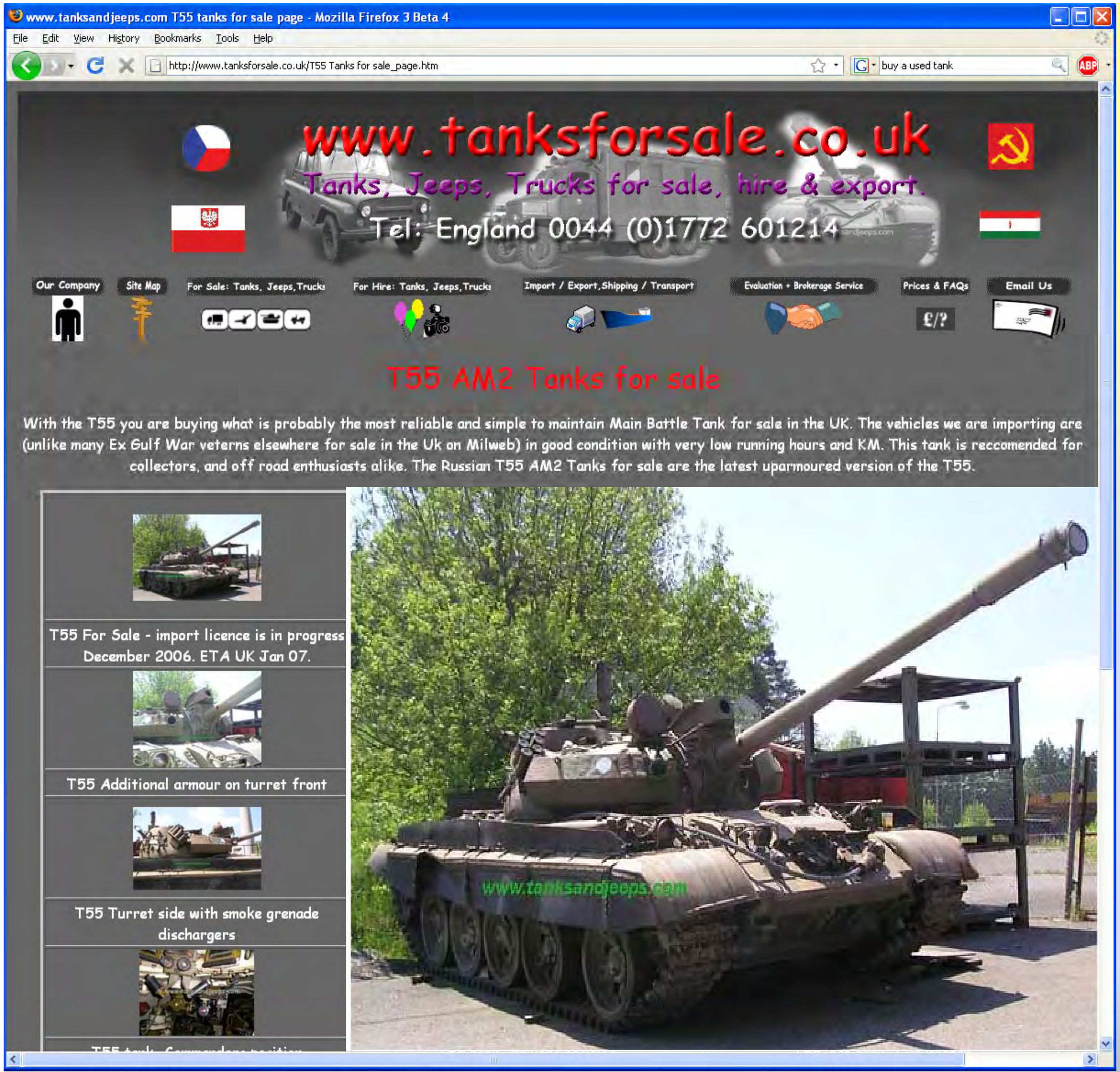
- that reduces its value to a favored stakeholder
- or increases its value to a disfavored stakeholder
- without a sufficiently large countervailing benefit.

An error:

- May or may not be a coding error
- May or may not be a functional error

Any threat to the value of the product to any stakeholder who matters.

James Bach


www.tanksandjeeps.com T55 tanks for sale page - Mozilla Firefox 3 Beta 4
File Edit View History Bookmarks Tools Help
http://www.tanksforsale.co.uk/T55 Tanks for sale_page.htm
buy a used tank
www.tanksforsale.co.uk
Tanks, Jeeps, Trucks for sale, hire & export.
Tel: England 0044 (0)1772 601214
Our Company
Site Map
For Sale: Tanks, Jeeps, Trucks
For Hire: Tanks, Jeeps, Trucks
Import / Export, Shipping / Transport
Evaluation + Brokerage Service
Prices & FAQs
Email Us

T55 AM2 Tanks for sale

With the T55 you are buying what is probably the most reliable and simple to maintain Main Battle Tank for sale in the UK. The vehicles we are importing are (unlike many Ex Gulf War veterns elsewhere for sale in the Uk on Milweb) in good condition with very low running hours and KM. This tank is reccomended for collectors, and off road enthusiasts alike. The Russian T55 AM2 Tanks for sale are the latest uparmoured version of the T55.

T55 For Sale - import licence is in progress December 2006. ETA UK Jan 07.

T55 Additional armour on turret front

T55 Turret side with smoke grenade dischargers

Not every limitation on value is a bug:

Effective bug reporting requires evaluation of the product's context (market, users, environment, etc.)

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

Verification

IF you have contracted for delivery of software, and the contract contains a complete and correct specification,

verification-oriented testing can answer the question,

Do we have to pay for this software?

Verification

Verification-oriented testing can answer the question:

Do we have to pay for this software?

But if…

- You're doing in-house development
- With evolving requirements (and therefore an incomplete and non-authoritative specification).

Verification only begins to address the critical question:

Will this software meet our needs?

Verification / Validation

In system testing,

The primary reason we do verification testing is to assist in validation.

Will this software meet our needs?

System testing (validation)

Designing system tests is like doing a requirements analysis. They rely on similar information but use it differently.

- The requirements analyst tries to foster agreement about the system to be built. The tester exploits disagreements to predict problems with the system.
- The tester doesn't have to reach conclusions or make recommendations about how the product should work. Her task is to expose credible concerns to the stakeholders.
- The tester doesn't have to make the product design tradeoffs. She exposes the consequences of those tradeoffs, especially unanticipated or more serious consequences than expected.
- The tester doesn't have to respect prior agreements. (Caution: testers who belabor the wrong issues lose credibility.)
- The system tester's work cannot be exhaustive, just useful.

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

Testing is always a search for information

- 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 product support costs
- 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 require different testing tools and strategies and will yield different tests, different test documentation and different test results.

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

We pick the technique that provides the best set of attributes, given the information objective and the context.

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.

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

Differences in emphasis on different test attributes

- **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

Test Techniques

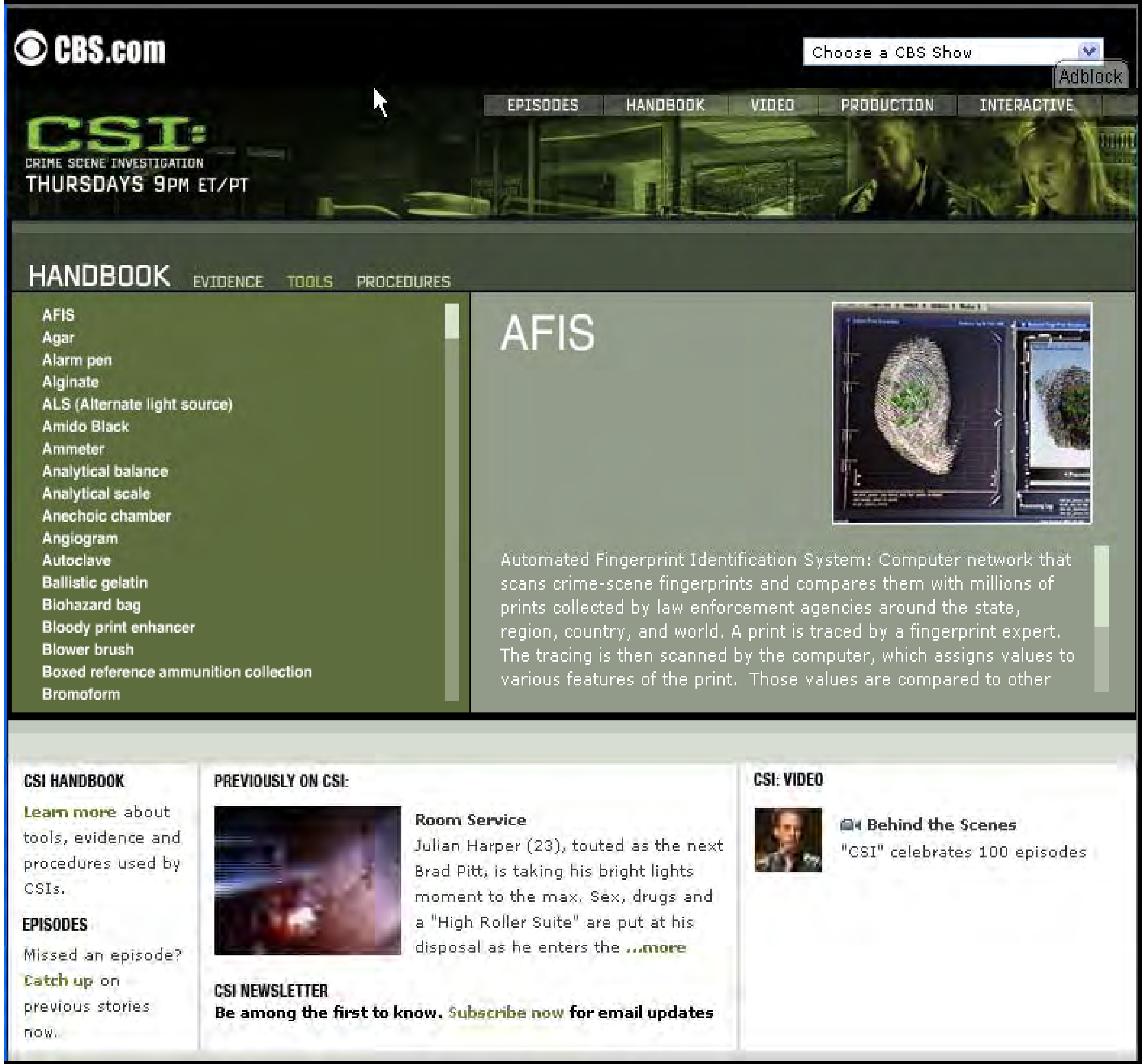
There might be as many as 150 named techniques

Different techniques are useful to different degrees in different contexts

Examples of important context factors

- Who are the stakeholders with influence
- What are the goals and quality criteria for the project
- What skills and resources are available to the project
- What is in the product
- How it could fail
- Potential consequences of potential failures
- Who might care about which consequence of what failure
- How to trigger a fault that generates a failure we're seeking
- How to recognize failure
- How to decide what result variables to attend to
- How to decide what *other* result variables to attend to in the event of intermittent failure
- How to troubleshoot and simplify a failure, so as to better
  - motivate a stakeholder who might advocate for a fix
  - enable a fixer to identify and stomp the bug more quickly
- How to expose, and who to expose to, undelivered benefits, unsatisfied implications, traps, and missed opportunities.

It's kind of like CSI



- AFIS
- Agar
- Alarm pen
- Alginate
- ALS (Alternate light source)
- Amido Black
- Ammeter
- Analytical balance
- Analytical scale
- Anechoic chamber
- Angiogram
- Autoclave
- Ballistic gelatin
- Biohazard bag
- Bloody print enhancer
- Blower brush
- Boxed reference ammunition collection
- Bromoform

AFIS

Automated Fingerprint Identification System: Computer network that scans crime-scene fingerprints and compares them with millions of prints collected by law enforcement agencies around the state, region, country, and world. A print is traced by a fingerprint expert. The tracing is then scanned by the computer, which assigns values to various features of the print. Those values are compared to other

CSI HANDBOOK

Learn more about tools, evidence and procedures used by CSIs.

EPISODES

Missed an episode? Catch up on previous stories now.

PREVIOUSLY ON CSI:

Room Service
Julian Harper (23), touted as the next Brad Pitt, is taking his bright lights moment to the max. Sex, drugs and a "High Roller Suite" are put at his disposal as he enters the ...more

CSI NEWSLETTER
Be among the first to know. Subscribe now for email updates

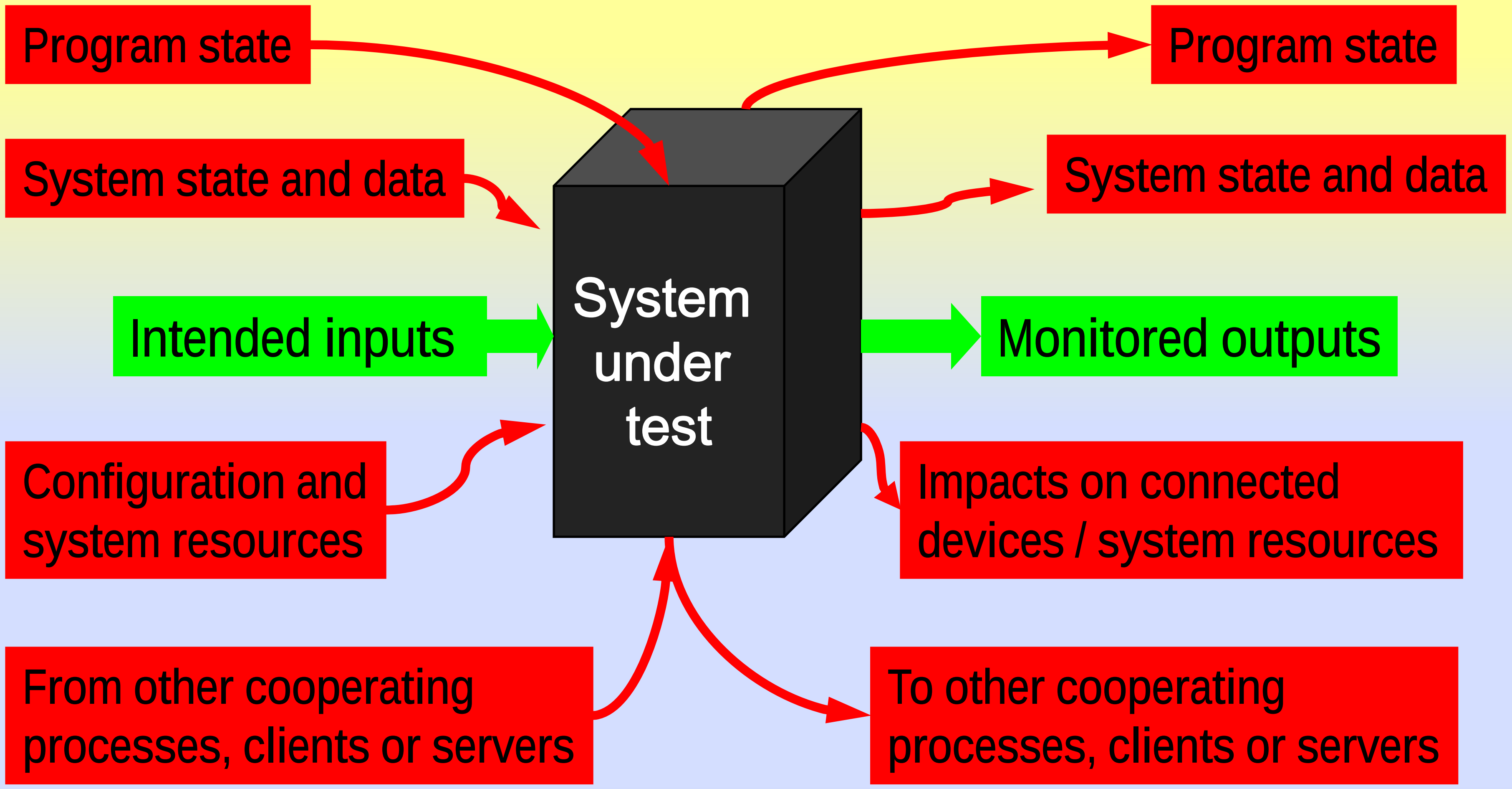
CSI: VIDEO

Behind the Scenes
"CSI" celebrates 100 episodes

MANY tools, procedures, sources of evidence.

- Tools and procedures don't define an investigation or its goals.
- There is too much evidence to test, tools are often expensive, so investigators must exercise judgment.
- The investigator must pick what to study, and how, in order to reveal the most needed information.

More on blind spots



Program state
System state and data
Intended inputs
Configuration and system resources
From other cooperating processes, clients or servers
System under test
Program state
System state and data
Monitored outputs
Impacts on connected devices / system resources
To other cooperating processes, clients or servers

Based on notes from Doug Hoffman

More on blind spots

The phenomenon of inattentional blindness

- humans (often) don't see what they don't pay attention to
- programs (always) don't see what they haven't been told to pay attention to

This is often the cause of irreproducible failures. We paid attention to the wrong conditions.

- But we can't pay attention to all the conditions

The 1100 embedded diagnostics

- Even if we coded checks for each of these, the side effects (data, resources, and timing) would provide us a new context for the Heisenberg principle

Our tests cannot practically address all of the possibilities

And *Even If* We Demonstrate a Failure That Doesn't Mean Anyone Will Fix It

The decision to fix a bug is rooted in a cost / benefit analysis

The quality of the bug description (the effectiveness of the tester) lies in its:

- Technical quality (scope and severity)
- Impact analysis (what are costs to who)
- Persuasiveness and clarity

System testers are bug advocates

- Client experienced a wave of serious product recalls (defective firmware)
  - Why were these serious bugs not found in testing?
    - ***They were found in testing and reported***
  - Why didn't the programmers fix them?
    - ***They didn't understand what they were reading***
  - What was wrong with the bug reports?
    - ***The problem is that the testers focused on creating reproducible failures, rather than on the quality of their communication.***
- Looking over 5 years of bug reports, I could predict fixes better by clarity/style/attitude of report than from severity

Bug advocacy = sales

- Time is in short supply.
- People are overcommitted.
- A bug report is a new task on an overcrowded to-do list.
- The art of motivating someone to do something that you want them to do is called sales.
- We can think of the sales communication in terms of:
  - Motivating the buyer
  - Overcoming the buyer's objections
- Serious failures might be motivating, but reports that are poorly worded or convey messages that cut credibility create objections—they get dismissed.

Sales = software engineering?

- Persuasive communication comes up in many software engineering contexts.
- Famous examples
  - Challenger disaster
  - David Parnas' warnings on SDI (Star Wars)
  - Electronic voting equipment
- Routine example
  - Status reporting in the face of unreasonable demands (Death March)
- But if we **study the communication as a software engineering** problem, how much traction does that give us?
- Maybe we gain more insight from thinking about human-to-human communications (like, sales).

Let's Sum Up

Is testing ONLY concerned with the human issues associated with product development and product use?

- Of course not
- But thinking in terms of the human issues leads us into interesting questions about
  - what tests we are running (and why)
  - what risks we are anticipating
  - how/why these risks are important, and
  - what we can do to help our clients get the information they need to manage the project, use the product, or interface with other professionals.

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.