Fundamental Challenges in Software Testing

Cem Kaner
Florida Tech

Colloquium Presentation at Butler University, April 2003

This research was partially supported by NSF Grant EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers." 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 (NSF).

What’s So Special About Testing?

- Wide array of issues: technical, psychological, project management, marketing, application domain.
- The rubber meets the road here
  - Toward the end of the project, there is little slack left. Decisions have impact *now*. The difficult decisions must be faced and made.
  - Testing plays a make-or-break role on the project.
    - An effective test manager and senior testers can facilitate the release of a high-quality product.
    - Less skilled testing staff create more discord than their technical contributions (such as they are) are worth.

Four Fundamental Challenges to Competent Testing

- Complete testing is impossible
- Testers misallocate resources because they fall for the company’s process myths
- Test groups operate under multiple missions, often conflicting, rarely articulated
- Test groups often lack skilled programmers, and a vision of appropriate projects that would keep programming testers challenged

1. Complete Testing is Impossible

- There are enormous numbers of possible tests. To test everything, you would have to:
  - Test every possible input to every variable.
  - Test every possible combination of inputs to every combination of variables.
  - Test every possible sequence through the program.
  - Test every hardware / software configuration, including configurations of servers not under your control.
  - Test every way in which the user might try to use the program.

1. Complete Testing is Impossible

The Problem of Coverage

- One approach to the problem has been to (attempt to) simplify it away, by saying that you achieve "complete testing" if you achieve "complete coverage".
- **What *is* coverage?**
  - Extent of testing of certain attributes or pieces of the program, such as statement coverage or branch coverage or condition coverage.
  - **Extent of testing completed, compared to a population of possible tests.**
- Typical definitions are oversimplified. They miss, for example,
  - Interrupts and other parallel operations
  - Interesting data values and data combinations
  - Missing code
- In practice, the number of variables we might measure is stunning. I listed 101 examples in *Software Negligence & Testing Coverage.*

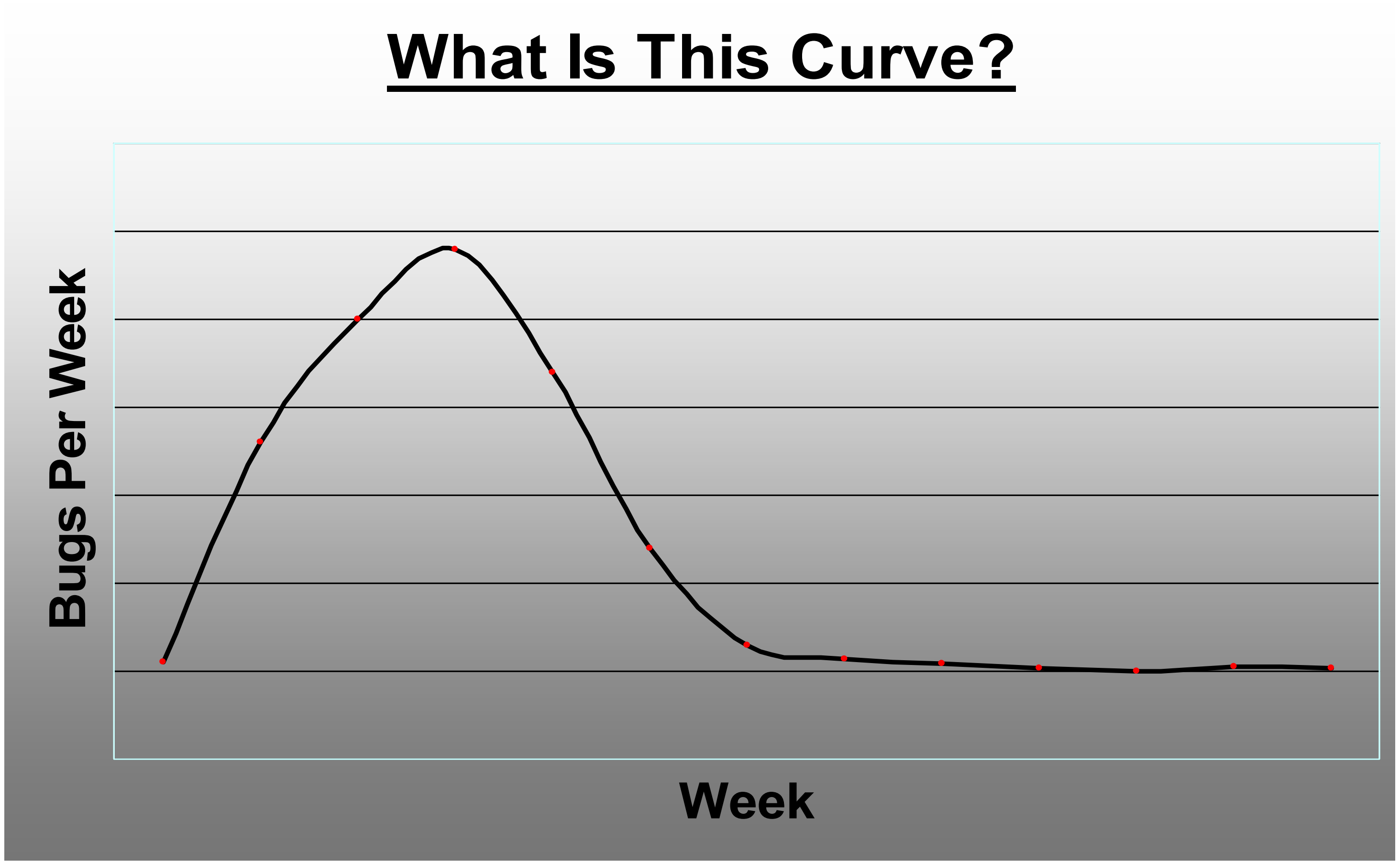
Measuring and Achieving High Coverage

- ***Coverage measurement is an interesting way to tell that you are far away from complete testing, but testing in order to achieve a "high" coverage is likely to result in development of a mass of low-power tests.***
  - People optimize what we measure them against, at the expense of what we don't measure.
  - Brian Marick, raises this and several other issues in his papers at www.testing.com (e.g. *How to Misuse Code Coverage*). Brian has been involved in development of several of the commercial coverage tools.

Can Bug Curves Tell Us When We’re Done ?

- Another way people measure completeness, or extent, of testing is by plotting bug curves, such as
  - New bugs found per week
  - Bugs still open (each week)
  - Ratio of bugs found to bugs fixed (per week)
- We fit the curve to a theoretical curve, often a probability distribution, and read our position from the curve. At some point, it is “clear” from the curve that we’re done.

The Bug Curve



A Common Model (Weibull) and its Assumptions

1. Testing occurs in a way that is similar to the way the software will be operated.
2. All defects are equally likely to be encountered.
3. All defects are independent.
4. There is a fixed, finite number of defects in the software at the start of testing.
5. The time to arrival of a defect follows the Weibull distribution.
6. 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.

The Weibull Distribution

- I think it is absurd to rely on a distributional 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 WHAT DOES THAT TELL US? HOW SHOULD WE INTERPRET IT?

1. Complete Testing is Impossible

Side Effects of Bug Curves

Earlier in testing: (Pressure is to increase bug counts)

- 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.)

- For more on measurement dysfunction, read Bob Austin's book, *Measurement and Management of Performance in Organizations*.

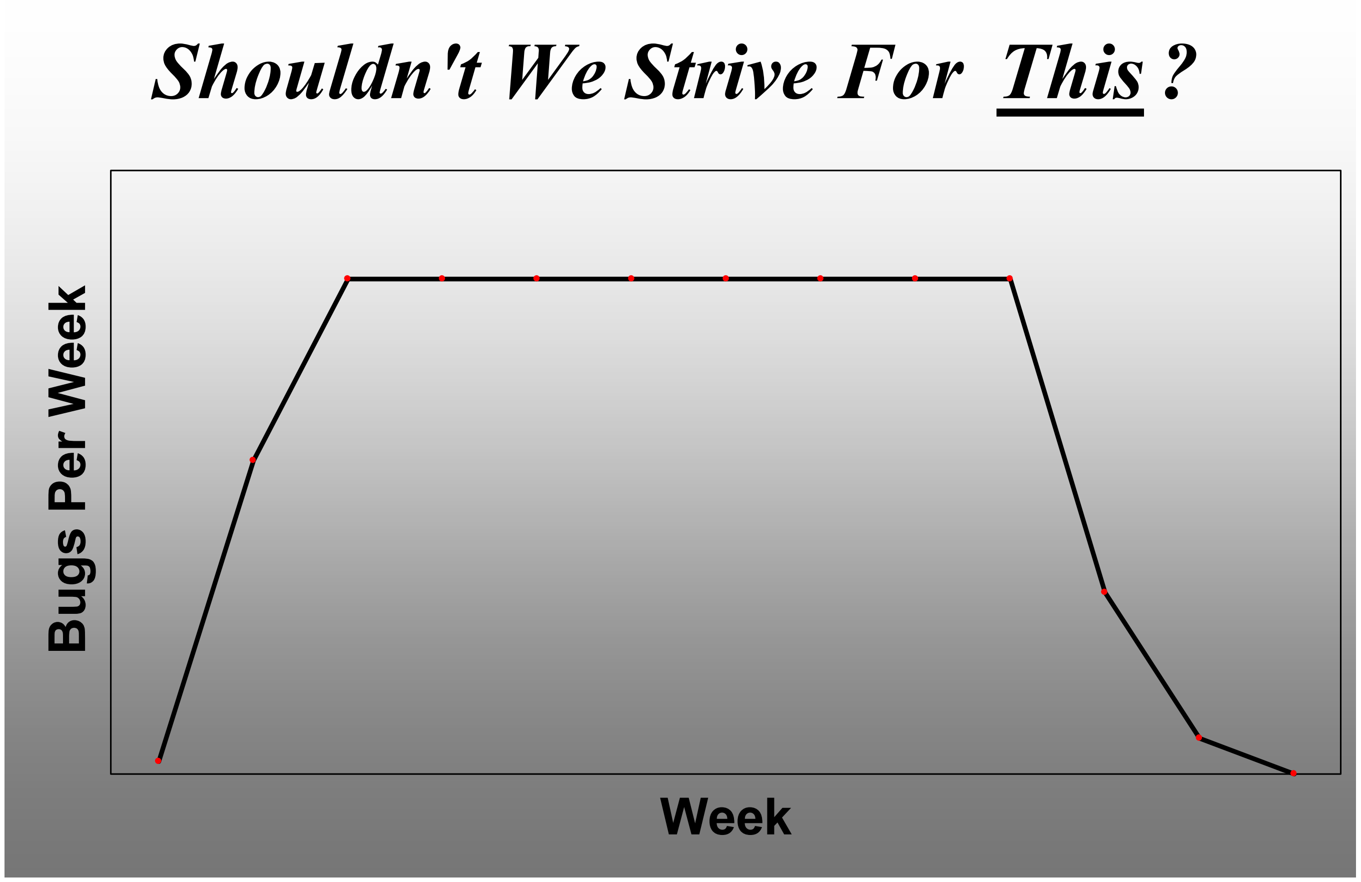
1. Complete Testing is Impossible

Side Effects of Bug Curves

Later in testing: *Pressure is to decrease 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

When you get past the simplistic answers, you realize that

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

Example: Time you spend on

- analyzing, troubleshooting, and effectively describing a failure

Is time no longer available for

- Designing tests	- Documenting tests
- Executing tests	- Automating tests
- Reviews, inspections	- Supporting tech support
- Retooling	- Training other staff

Testers Live and Breathe Tradeoffs

Some

- standards,
- texts
- luminaries

make absolute statements. You must always do task X, or if you do task Y, it must always contain these components.

These inspire guilt, but they don't provide useful guidance.

Example: IEEE Standard 829 for software test documentation seems to be liked in medical or aerospace-related companies, but it has probably done more harm than good in most commercial situations.

There are too many important tasks for testers to do. We have to mature our judgment in order to decide which of these *not* to do or to do only lightly.

Read Drucker's, *The Effective Executive.*

Even More Tradeoffs

- From an infinitely large population of tests, we can only run a few. Which few do we select?
- Competing characteristics of good tests. One test is better than another if it is:
  - More powerful
  - More likely to yield significant (more motivating, more persuasive) results
  - More credible
  - Representative of events more likely to be encountered by the user
  - Easier to evaluate.
  - More useful for troubleshooting
  - More informative
  - More appropriately complex
  - More likely to help the tester or the programmer develop insight into some aspect of the product, the customer, or the environment
- ***No test satisfies all of these characteristics. How do we balance them?***

Four Fundamental Challenges to Competent Testing

- Complete testing is impossible
  - *There is no simple answer for this.*
  - *Therefore testers live and breathe tradeoffs.*
- **Testers misallocate resources because they fall for the company's process myths**
- Test groups operate under multiple missions, often conflicting, rarely articulated
- Test groups often lack skilled programmers, and a vision of appropriate projects that would keep programming testers challenged

2. Process myths

You Can Trust Me on *This*

- We follow the waterfall lifecycle
- We collect all of the product requirements at the start of the project, and we can rely on the requirements document throughout the project.
- We write thorough, correct specifications and keep them up to date.
- The customer will accept a program whose behavior exactly matches the specification.
- We fix every bug of severity (or priority) level X and we never lower the severity level to avoid having to fix the bug.

Amazingly, many testers believe statements like this, Project after Project, and rely on them, Project after Project.

Effects of Relying on Process Myths

- Testers design their tests from the specs / requirements, long before they get the code. After all, we know what the program will be like.
- Testers evaluate program capability in terms of conformance to the written requirements, suspending their own judgment. After all, we know what the customer wants.
- Testers evaluate program correctness only in terms of conformance to specification, suspending their own judgment. After all, this is what the customer wants.
- Testers build extensive, fragile, GUI-level regression test suites. After all, the UI is fully specified. We know it's not going to change.

Multiple Missions, Rarely Articulated

- Find defects
- Block premature product releases
- Help managers make ship / no-ship decisions
- Minimize technical support costs
- Assess conformance to specification
- Conform to regulations
- Minimize safety-related lawsuit risk
- Find safe scenarios for use of the product
- Assess quality
- Verify correctness of the product
- Assure quality
- ***Do you know what your group's mission is? Does everyone in your company agree?***

3. Multiple Missions

One Example: The Agency Problem

- Products are designed and built by and for ***multiple stakeholders***
  - They have conflicting interests, needs and preferences
  - The requirements analyst and programming team ***seek to resolve the differences*** among the stakeholders
  - <<Implicit Mission>> Testers identify issues that will dissatisfy individual stakeholders.
  - We advocate for bug fixes by appealing to specific stakeholders who will be more affected by the problems. ***We often surface and reinforce the differences among stakeholders.***
- **Why would we do this?**

Four Fundamental Challenges to Competent Testing

- Complete testing is impossible
  - *There is no simple answer for this. Therefore testers live and breathe tradeoffs.*
- Testers misallocate resources because they fall for the company's process myths.
  - *Testers have to rely on their wits, not on someone else's compliance with an (alleged but unrealistic) process.*
- Test groups operate under multiple missions, often conflicting, rarely articulated.
  - *We pick our tests to conform to our testing mission.*
- **Test groups often lack skilled programmers, and a vision of appropriate projects that would keep programming testers challenged.**

Causes

- The optimal test group has diversity of skills and knowledge (see next slide). This is easily misunderstood:
  - Weakness in programming skill is seen as weakness in testing skill (and vice-versa).
  - Strength in programming is seen as assuring strength in testing.
- Many common testing practices do not require programming knowledge or skill.
- People who want to be in Product Development but who can't code have nowhere else to go.
- People who are skilled programmers are afraid of dead-ending in a test group.

4. Weak Programmers

Causes—Breadth of Needed Skills & Knowledge

- To name a few,
  - Testing techniques
  - Strong communications
  - Application level programming
  - System level programming
  - Various types of devices (e.g. for printers, knowledge of market shares, diagnostics, drivers, configuration, risks for each device, etc.)
  - All aspects of the software under test
  - Mathematics, especially combinatorics and probability theory.
  - Project management
  - Project accounting
  - Failure analysis
  - Products liability and contract liability laws, etc.
- ***You can't find a person with all these skills and areas of knowledge. The trick is to build a group of specialists who cross-train each other.***

4. Weak Programmers

Causes—Different Styles of Black Box Testing

- Function testing
- Domain testing
- Specification-based testing
- Risk-based testing
- Stress testing
- Regression testing
- User testing
- Scenario testing
- State-model based testing
- High volume automated testing
- Exploratory testing

Few of these require programming skill and none requires knowledge of internals of the program.

4. Weak Programmers

Effects: Overbalance of Process vs. Technical Analysis

What's wrong with process?

- Controversy in the definition of the undergrad software engineering degree.
- IEEE/ACM SEEK committee seems determined to push multiple courses on software process and software management into the undergraduate curriculum.
  - Academics see undergraduate projects, in which students arrogantly dispense with all process and produce mediocre work, slowly.
  - We've all seen (personally or in the books) large projects fail because of obvious process failures.

Maybe we need <u>more</u> attention to process?

4. Weak Programmers

Effects: Overbalance of Process vs. Technical Analysis

It Looks Different when You Look at Many Test Groups.

- People who have never been, and never will become, project managers constantly push processes that "They" (project mgr and development staff) should follow.
- Process advocates push reliance on standards that might work well for large military projects but are absurd for many commercial projects. Testing templates based on IEEE 829 are particularly common and particularly wasteful.
- Process advocates spend enormous amount of time lobbying on process / political issues, getting little technical work done and often negatively impacting morale.

4. Weak Programmers

Effects: Overbalance of Process vs. Technical Analysis

Do you read Dilbert?

- We need more technically sophisticated, smarter Ratberts
- Not more process improvement specialists who can't test their way out of a loop but who always know how *"They"* could do it better.

4. Weak Programmers

Effects: Overbalance of Process vs. Technical Analysis

To Avoid Misunderstanding

- I agree that good processes are important.
  - *Testers should improve the effectiveness of testing processes.*
- I agree that there are skilled process consultants, and that these folks can add value to the business.
  - *Testers have no special charter to serve as general process consultants.*

Effects: The GUI Regression Testing Paradigm

This is the most commonly discussed automation approach:

- create a test case
- run it and inspect the output
- if the program fails, report a bug and try again later
- if the program passes the test, save the resulting outputs
- in future tests, run the program and compare the output to the saved results. Report an exception whenever the current output and the saved output don’t match.

4. Weak Programmers

Effects: Is this Really Automation?

Analyze product	--	human
Design test	--	human
Run test 1st time	--	human
Evaluate results	--	human
Report 1st bug	--	human
Save code	--	human
Save result	--	human
Document test	--	human
Re-run the test	--	MACHINE
Evaluate result	--	machine *plus human if there's any mismatch*
Maintain result	--	human

Woo-hoo! We really get the machine to do a *whole lot* of our work!

(Maybe, but not this way.)

4. Weak Programmers

Effects: Is GUI Automation Cost Effective?

- Test case creation is expensive. Estimates run from 3-5 times the time to create and manually execute a test case (Bender) to 3-10 times (Kaner) to 10 times (Pettichord) or higher (LAWST).
- You usually have to increase the testing staff in order to generate automated tests. Otherwise, how will you achieve the same breadth of testing?
- Your most technically skilled staff are tied up in automation.
- Automation can delay testing, adding even more cost (albeit hidden cost.)
- Excessive reliance leads to the 20 questions problem. (Fully defining a test suite in advance, before you know the program's weaknesses, is like playing 20 questions where you have to ask all the questions before you get your first answer.)

4. Weak Programmers

Effects: GUI Automation Pays off Late

- Regression testing has low power.
  - Run the test, program passes it. What is the probability that the program will fail later in this release?
    - Variable results (source control problems, source availability problems, fragile code, etc.) but the LAWST estimates were that about 12-15% of the bugs found across a wide range of projects were found with GUI regression tests.
    - This percentage is far less than the cost of creating and maintaining the tests.
  - Rerunning old tests that the program has passed is less powerful than running new tests.
- **Our main payback is usually in the next release, not this one.**
- Maintainability is, therefore, a core issue.

For more, see Kaner, Avoiding Shelfware: A Manager's View of Automated GUI Testing

4. Weak Programmers

Effects: Test Automation is Programming

Win NT 4 had 6 million lines of code, and 12 million lines of test code

Common (and often vendor-recommended) design and programming practices for automated testing are appalling:

- **Embedded constants**
- No modularity
- ***No source control***
- No documentation
- No requirements analysis
- No wonder we fail.
- And no wonder no self-respecting programmer wants to join a test group to write this kind of code.

4. Weak Programmers

Effects: Intellectual Stagnation

- British Computer Society Information Systems Examinations Board, Practitioner Certificate in Software Testing, Guidelines and Syllabus, September 2001. http://www1.bcs.org.uk/DocsRepository/00900/913/docs/practsyll.pdf
- ***This is one of the more reputable certification exams for software testers.***
  - "The following are the pre-requisites for test execution: Test procedure and/or test script."
  - Test execution "is a comparison of actual and expected outcomes, and [that] the expected outcome must be generated prior to test execution"
  - "Every test case is logged in the test record for the purpose of auditing the testing, and recording test coverage measures for subsequent checking against test completion criteria."
- Most of this document could have been written in 1980, and I would have considered much of it outdated or inapplicable back in 1984, when I started writing *Testing Computer Software.*

New Directions: XP-inspired Collaboration

- New collaboration model to support better development and maintenance:
  - junit, cppunit, testunit, etc. -- www.junit.org
  - FIT – http://fit.c2.com
  - Many other tools, see Pettichord's *Homebrew Test Automation* www.io.com/~wazmo/papers/home_brew_test_automation_20030312.pdf

- Open source automation tools, including regression automation at the unit level and the API level
  - Failures immediately visible to the programmer
  - Many side-effects have immediately visible effect
  - Tests are unperturbed by change at the UI level and by many other changes in functionality

- Test-driven development

New Directions: High Volume Test Automation

- Massive number of thematically related tests
  - Human designs the overall test strategy (and writes the appropriate code), but then the computer designs, executes and evaluates the tests, calling for human intervention only when needed.
- Examples
  - Exhaustive or large sample random testing against partial and heuristic oracles
  - State-model based testing
  - Simulator-based testing using probes
  - Random sequences of pre-passed tests
- See Kaner, Architectures of Test Automation, http://www.kaner.com/testarch.html

New Directions: Higher Skill Manual Testing

- Exploratory Testing
  - Testing is an active learning effort (simultaneous test design, learning, and test execution).
  - We’re studying the cognitive psychology of the brain-engaged tester (see James Bach’s methodology papers at www.satisfice.com; look for additional papers by Andy Tinkham, a doctoral student in my lab).