

Black Box Software Testing

Part 10.

Black Box Testing Paradigms

ACKNOWLEDGEMENT

This section is based on work done jointly by Cem Kaner and James Bach. Bob Stahl, Brian Marick, Hans Schaefer, and Hans Buwalda also provided several insights.

Copyright Notice

These slides are distributed under the Creative Commons License.

In brief summary, you may make and distribute copies of these slides so long as you give the original author credit and, if you alter, transform or build upon this work, you distribute the resulting work only under a license identical to this one.

For the rest of the details of the license, see <http://creativecommons.org/licenses/by-sa/2.0/legalcode>.

The Puzzle

Black box testing groups vary widely in their approach to testing.

Tests that seem essential to one group seem uninteresting or irrelevant to another.

Big differences can appear even when both groups are composed of intelligent, experienced, dedicated people.

Why?

A List of Testing Paradigms

- **Domain**
- **Function**
- **Regression**
- **Specification-based**
- **User**
- **Scenario**
- **Risk-based**
- **Stress**
- **High volume stochastic or random**
- **State model based**
- **Exploratory**

Domain Testing

Key Idea: *“Divide and conquer the data”*

Summary:

- Look for any data processed by the product. Look at outputs as well as inputs.
- Decide which data to test with. *Consider things like boundary values, typical values, convenient values, invalid values, or best representatives.*
- Consider combinations of data worth testing together.

Good for: all purposes

Function Testing

Key Idea: *“Test what it can do”*

Summary:

- A function is something the product can do.
- Identify each function and sub-function.
- Determine how you would know if they worked.
- Test each function, one at a time.
- See that each function does what it's supposed to do, and not what it isn't.

Good for: assessing capability rather than reliability

Regression Testing

Key Idea: *“Repeat testing after changes.”*

- Summary:**
- Build a suite of tests
 - Run the tests when anything changes
 - Bug regression (Show that a bug was not fixed)
 - Old fix regression (Show that an old bug fix was broken)
 - General functional regression (Show that a change caused a working area to break.)

Good for: Building confidence

Specification Based Testing

Key Idea: *“Verify every claim”*

Summary:

- Identify specifications (implicit or explicit).
- Analyze individual claims about the product.
- Work to clarify vague claims.
- Verify that each claim about the product is true.
- Expect the specification and product to be brought into alignment.

Good for: simultaneously testing the product and specification, while refining expectations

User Testing

Key Idea: *“Involve the users”*

Summary:

- Identify categories and roles of users.
- Determine what each category of user will do, how they will do it, and what they value.
- Get real user data, or bring real users in to test.
- Otherwise, systematically simulate a user.
- Powerful user testing is that which involves a variety of users and user roles, not just one.

Good for: all purposes

Scenario Testing

Key Idea: *“Do one thing after another”*

Summary:

- Define test procedures or high level cases that incorporate multiple activities connected end to end.
- Don't reset the system between events.
- Can vary timing and sequencing, and try parallel threads.

Good for: finding problems fast
(however, bug analysis is more difficult)

Risk Based Testing

Key Idea: *“Imagine a problem, then look for it.”*

Summary:

- What kinds of problems could the product have?
- Which problems matter most? Focus on those.
- How would you detect them if they were there?
- Make a list of interesting problems and design tests specifically to reveal them.
- It may help to consult experts, design documentation, past bug reports, or apply risk heuristics.

Good for: making best use of testing resources;
leveraging experience

Stress Testing

Key Idea: *“Overwhelm the product”*

Summary:

- Look for functions or sub-systems of the product that may be vulnerable to failure due to challenging input or constrained resources.
- Identify input or resources related to those functions or sub-systems.
- Select or generate challenging data and platform configurations to test with: e.g., *large or complex data structures, high loads, long test runs, many test cases, limited memory, etc.*

Good for: performance, reliability, and efficiency assessment

Volume Random Testing

Key Idea: *“Run a million different tests”*

Summary:

- Look for an opportunity to automatically generate thousands of slightly different tests.
- Create an automated, high speed oracle.
- Write a program to generate, execute, and evaluate all the tests.

Good for: Assessing reliability across input and time.

State Model Testing

Key Idea: *“Use a computer model of the SUT as test input.”*

Summary:

- Describe a model of the SUT.
- Create a machine readable form of the model.
- Write a program to use the model as a basis for testing.

Good for: Stable, well defined state machines.

Exploratory Testing

Key Idea: *“Simultaneous learning, test design, and test execution.”*

Summary:

- Plan a testing mission.
- Learn about the product as you test.
- Document observations, problems, and plans as you go.

Good for: Early testing, high risk problems.

Paradigm Exercise

- **Do any of the paradigms listed reflect a dominant approach in your company? Which one(s)?**
- **Looking at the paradigms as styles of testing, which styles are in use in your company? (List them from most common to least.)**
- **Of the ones that are not common or not in use in your company, is there one that looks useful, that you think you could add to your company's repertoire?**

Black Box Software Testing

Paradigms:

Domain Testing

Domain Testing

AKA partitioning, equivalence analysis, boundary analysis

Fundamental question or goal:

- This confronts the problem that there are too many test cases for anyone to run. This is a stratified sampling strategy that provides a rationale for selecting a few test cases from a huge population.

General approach:

- Divide the set of possible values of a field into subsets, pick values to represent each subset. Typical values will be at boundaries. More generally, the goal is to find a “best representative” for each subset, and to run tests with these representatives.
- Advanced approach: combine tests of several “best representatives”. Several approaches to choosing optimal small set of combinations.

Paradigmatic case(s)

- Equivalence analysis of a simple numeric field.
- Printer compatibility testing (*multidimensional variable, doesn't map to a simple numeric field, but stratified sampling is essential.*)

Domain Testing

Strengths

- Find highest probability errors with a relatively small set of tests.
- Intuitively clear approach, generalizes well

Blind spots

- Errors that are not at boundaries or in obvious special cases.
- Also, the actual domains are often unknowable.

Domain Testing

Some Key Tasks

- Partitioning into equivalence classes
- Discovering best representatives of the sub-classes
- Combining tests of several fields
- Create boundary charts
- Find fields / variables / environmental conditions
- Identify constraints (non-independence) in the relationships among variables.

Domain Testing

Some Relevant Skills

- Identify ambiguities in specifications or descriptions of fields
- Find biggest / smallest values of a field
- Discover common and distinguishing characteristics of multi-dimensional fields, that would justify classifying some values as “equivalent” to each other and different from other groups of values.
- Standard variable combination methods, such as all-pairs or the approaches in Jorgensen and Beizer’s books

Domain Testing: Interesting Papers

- Thomas Ostrand & Mark Balcer, *The Category-partition Method For Specifying And Generating Functional Tests*, Communications of the ACM, Vol. 31, No. 6, 1988.
- Debra Richardson, et al., *A Close Look at Domain Testing*, IEEE Transactions On Software Engineering, Vol. SE-8, NO. 4, July 1982
- Michael Deck and James Whittaker, *Lessons learned from fifteen years of cleanroom testing*. STAR '97 Proceedings (in this paper, the authors adopt boundary testing as an adjunct to random sampling.)
- Richard Hamlet & Ross Taylor, *Partition Testing Does Not Inspire Confidence*, Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, IEEE Computer Society Press, 206-215, July 1988

Domain Testing: Another Paper of Interest

Partition Testing Does Not Inspire Confidence, Hamlet, Richard G. and Taylor, Ross, Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, IEEE Computer Society Press, 206-215, July 1988

abstract = { Partition testing, in which a program's input domain is divided according to some rule and test conducted within the subdomains, enjoys a good reputation. However, comparison between testing that observes partition boundaries and random sampling that ignores the partitions gives the counterintuitive result that partitions are of little value. In this paper we improve the negative results published about partition testing, and try to reconcile them with its intuitive value. Partition testing is show to be more valuable than random testing only when the partitions are narrowly based on expected faults and there is a good chance of failure. For gaining confidence from successful tests, partition testing as usually practiced has little value.}

From the STORM search page:

<http://www.mtsu.edu/~storm/bibsearch.html>

Black Box Software Testing

Paradigms:

Function Testing

Function Testing

Tag line

- “Black box unit testing.”

Fundamental question or goal

- Test each function thoroughly, one at a time.

Paradigmatic case(s)

- Spreadsheet, test each item in isolation.
- Database, test each report in isolation

Strengths

- Thorough analysis of each item tested

Blind spots

- Misses interactions, misses exploration of the benefits offered by the program.

Some Function Testing Tasks

Identify the program's features / commands

- From specifications or the draft user manual
- From walking through the user interface
- From trying commands at the command line
- From searching the program or resource files for command names

Identify variables used by the functions and test their boundaries.

Identify environmental variables that may constrain the function under test.

Use each function in a mainstream way (positive testing) and push it in as many ways as possible, as hard as possible.

Black Box Software Testing

Paradigms:

Regression Testing

Regression Testing

Tag line

- “Repeat testing after changes.”

Fundamental question or goal

- Manage the risks that (a) a bug fix didn't fix the bug or (b) the fix (or other change) had a side effect.

Paradigmatic case(s)

- Bug regression (Show that a bug was not fixed)
- Old fix regression (Show that an old bug fix was broken)
- General functional regression (Show that a change caused a working area to break.)
- Automated GUI regression suites

Strengths

- Reassuring, confidence building, regulator-friendly

Regression Testing

Blind spots / weaknesses

- Anything not covered in the regression series.
- Repeating the same tests means not looking for the bugs that can be found by other tests.
- Pesticide paradox
- Low yield from automated regression tests
- Maintenance of this standard list can be costly and distracting from the search for defects.

Automating Regression Testing

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.

Potential Regression Advantages

- Dominant paradigm for automated testing.
- Straightforward
- Same approach for all tests
- Relatively fast implementation
- Variations may be easy
- Repeatable tests

The GUI Regression Automation Problem

Prone to failure because of difficult financing, architectural, and maintenance issues.

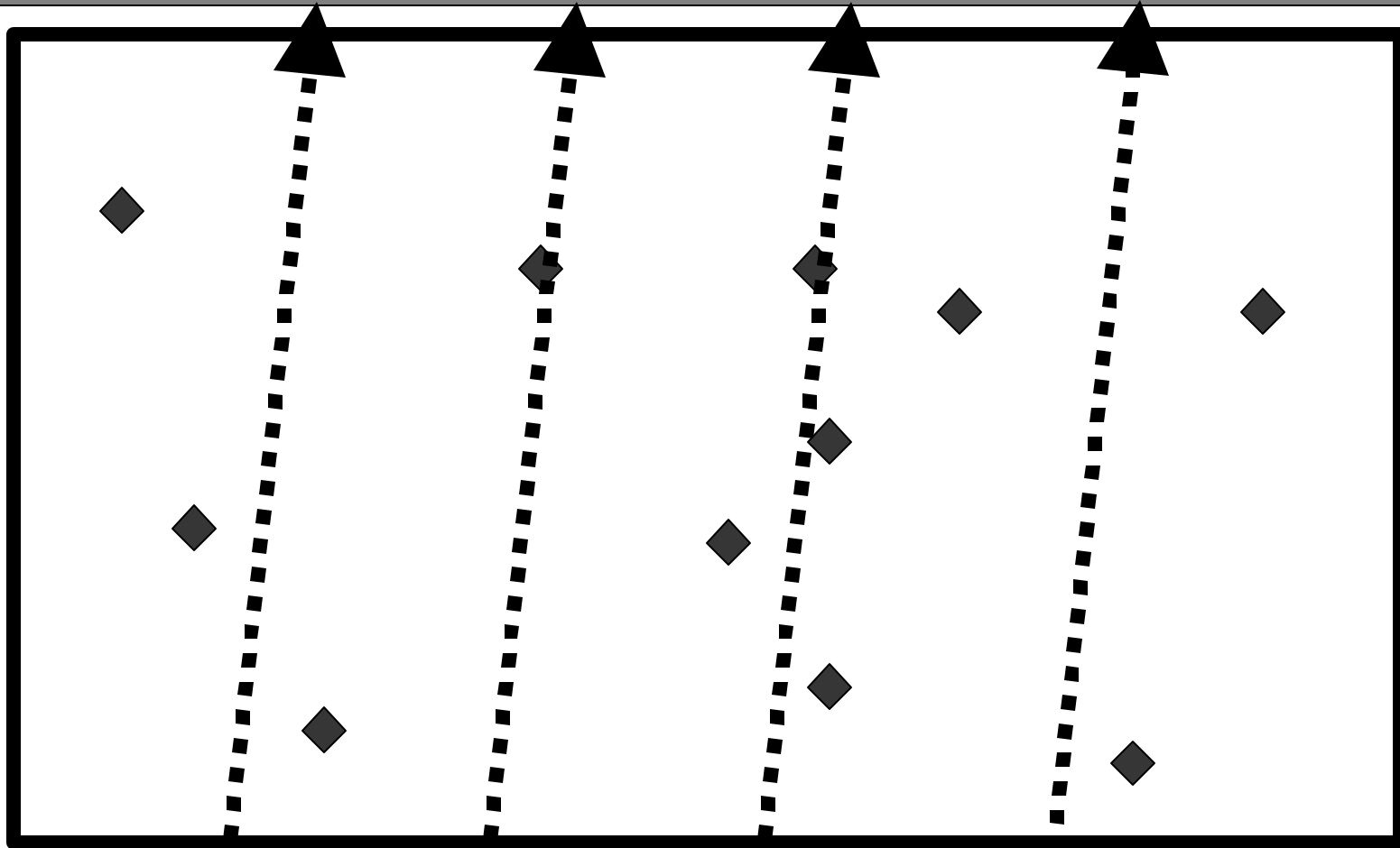
Low power (in its traditional form) even if successful.

Extremely valuable under some circumstances.

***THERE ARE MANY ALTERNATIVES THAT CAN BE
MORE APPROPRIATE UNDER OTHER
CIRCUMSTANCES.***

If your only tool is a hammer, everything looks like a nail.

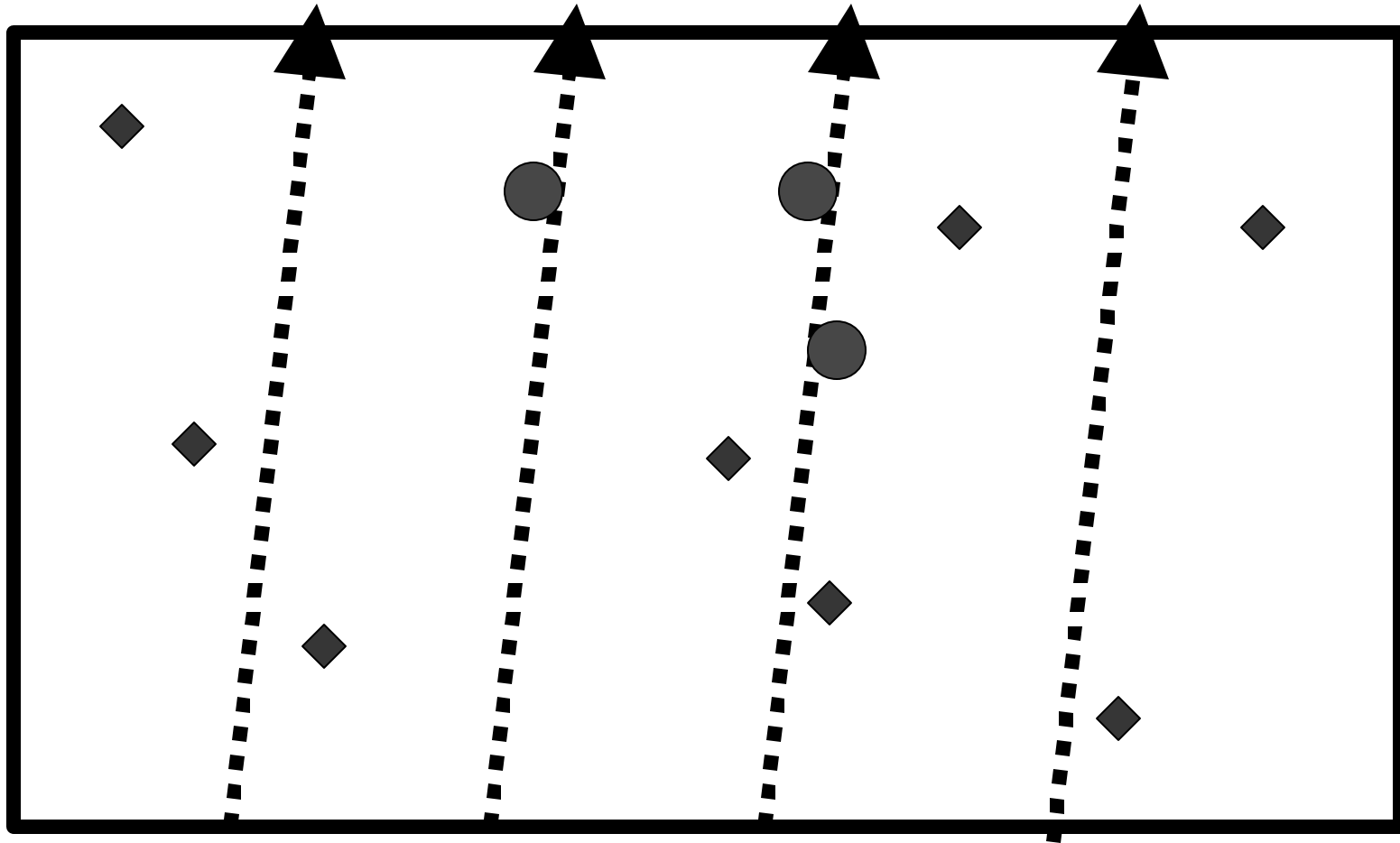
Testing Analogy: Clearing Mines



◆ mines

This analogy was first presented by Brian Marick.
These slides are from James Bach..

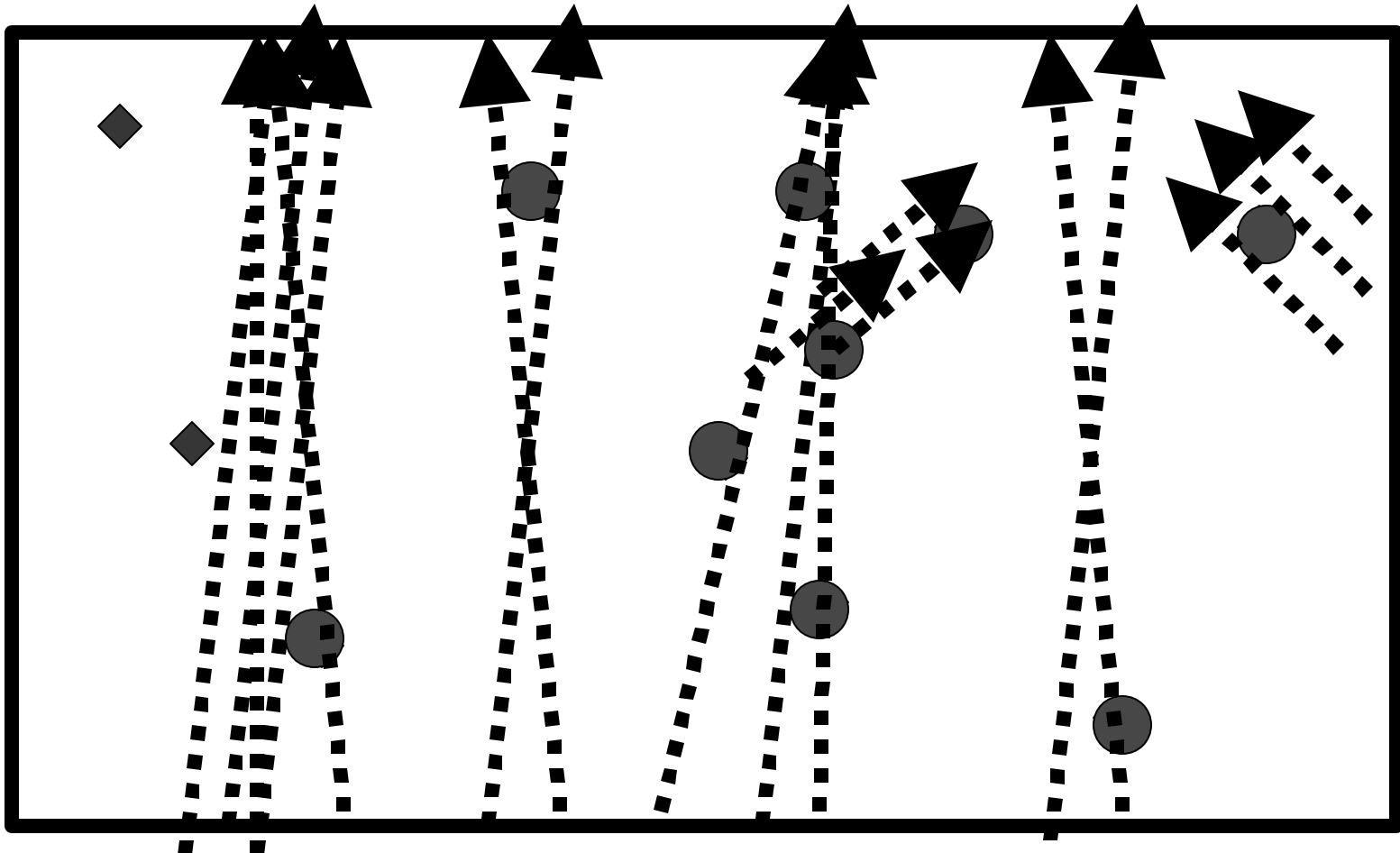
Totally Repeatable Tests Won't Clear the Minefield



◆ mines

● fixes

Variable Tests are Often More Effective



◆ mines ● fixes

GUI Regression Strategies: Some Papers of Interest

- Chris Agruss, *Automating Software Installation Testing*
- James Bach, *Test Automation Snake Oil*
- Hans Buwalda, *Testing Using Action Words*
- Hans Buwalda, *Automated testing with Action Words: Abandoning Record & Playback*
- Elisabeth Hendrickson, *The Difference between Test Automation Failure and Success*
- Cem Kaner, *Avoiding Shelfware: A Manager's View of Automated GUI Testing*
- John Kent, *Advanced Automated Testing Architectures*
- Bret Pettichord, *Success with Test Automation*
- Bret Pettichord, *Seven Steps to Test Automation Success*
- Keith Zambelich, *Totally Data-Driven Automated Testing*

Black Box Software Testing

Paradigms:

Specification-Based Testing

Specification-Driven Testing

Tag line:

- “Verify every claim.”

Fundamental question or goal

- Check the product’s conformance with every statement in every spec, requirements document, etc.

Paradigmatic case(s)

- Traceability matrix, tracks test cases associated with each specification item.
- User documentation testing

Specification-Driven Testing

Strengths

- Critical defense against warranty claims, fraud charges, loss of credibility with customers.
- Effective for managing scope / expectations of regulatory-driven testing
- Reduces support costs / customer complaints by ensuring that no false or misleading representations are made to customers.

Blind spots

- Any issues not in the specs or treated badly in the specs /documentation.

Traceability Matrix

	Var 1	Var 2	Var 3	Var 4	Var 5
Test 1	X	X	X		
Test 2		X		X	
Test 3	X		X	X	
Test 4			X	X	
Test 5				X	X
Test 6	X				X

Var can be anything identified as needing testing
(e.g., a feature, input, or result)

Traceability Matrix

The columns involve different test items. A test item might be a function, a variable, an assertion in a specification or requirements document, a device that must be tested, any item that must be shown to have been tested.

The rows are test cases.

The cells show which test case tests which items.

If a feature changes, you can quickly see which tests must be reanalyzed, probably rewritten.

In general, you can trace back from a given item of interest to the tests that cover it.

This doesn't specify the tests, it merely maps their coverage.

Specification

Tasks

- review specifications for
 - » Ambiguity
 - » Adequacy (it covers the issues)
 - » Correctness (it describes the program)
 - » Content (not a source of design errors)
 - » Testability support
- Create traceability matrices
- Document management (spec versions, file comparison utilities for comparing two spec versions, etc.)
- Participate in review meetings

Specification

Skills

- Understand the level of generality called for when testing a spec item. For example, imagine a field X:
 - » We could test a single use of X
 - » Or we could partition possible values of X and test boundary values
 - » Or we could test X in various scenarios
 - » Which is the right one?
- Ambiguity analysis
 - » Richard Bender teaches this well. If you can't take his course, you can find notes based on his work in Rodney Wilson's **Software RX: Secrets of Engineering Quality Software**

Specification

Skills

- Ambiguity analysis
 - » Another book provides an excellent introduction to the ways in which statements can be ambiguous and provides lots of sample exercises: Cecile Cyrul Spector, *Saying One Thing, Meaning Another : Activities for Clarifying Ambiguous Language*

Breaking Statements into Elements

Make / read a statement about the program

Work through the statement one word at a time, asking what each word means or implies.

- *Thinkertoys* describes this as “slice and dice.”
- *Gause & Weinberg* develop related approaches as
 - » “Mary had a little lamb” (read the statement several times, emphasizing a different word each time and asking what the statement means, read that way)
 - » “Mary conned the trader” (for each word in the statement, substitute a wide range of synonyms and review the resulting meaning of the statement.)
 - » These approaches can help you ferret out ambiguity in the definition of the product. By seeing how different people could interpret a key statement (e.g. spec statement that defines part of the product), you can see new test cases to check which meaning is operative in the program.

Breaking Statements into Elements: An Example

Quality is value to some person

- Quality
 - »
 - »
 - »
 - Value
 - »
 - »
 - »
 - Some
 - »
 - »
 - »
 - Person
 - »
- *Who is this person?*
 - *How are you the agent for this person?*
 - *How are you going to find out what this person wants?*
 - *How will you report results back to this person?*
 - *How will you take action if this person is mentally absent?*

Reviewing a Specification for Completeness

Reading a spec linearly is not a particularly effective way to read the document. It's too easy to overlook key missing issues.

We don't have time to walk through this method in this class, but the general approach that I use is based on James Bach's "Satisfice Heuristic Test Strategy Model" at <http://www.satisfice.com/tools/satisfice-tsm-4p.pdf>.

- You can assume (not always correctly, but usually) that every sentence in the spec is meant to convey information.
- The information will probably be about
 - » the project and how it is structured, funded or timed, or
 - » about the product (what it is and how it works) or
 - » about the quality criteria that you should evaluate the product against.

Reviewing a Specification for Completeness

Spec Review using the Satisfice Model, continued

- The Satisfice Model lists several examples of project factors, product elements and quality criteria.
- For a given sentence in the spec, ask whether it is telling you project, product, or quality-related information. Then ask whether you are getting the full story. As you do the review, you'll discover that project factors are missing (such as deadline dates, location of key files, etc.) or that you don't understand / recognize certain product elements, or that you don't know how to tell whether the program will satisfy a given quality criterion.
- Write down these issues. These are primary material for asking the programmer or product manager about the spec.

Getting Information When There Is No Specification

(Suggestions from some brainstorming sessions.)

- Whatever specs exist
- Software change memos that come with each new internal version of the program
- User manual draft (and previous version's manual)
- Product literature
- Published style guide and UI standards
- Published standards (such as C-language)
- 3rd party product compatibility test suites
- Published regulations
- Internal memos (e.g. project mgr. to engineers, describing the feature definitions)
- Marketing presentations, selling the concept of the product to management
- Bug reports (responses to them)
- Reverse engineer the program.
- Interview people, such as
 - development lead
 - tech writer
 - customer service
 - subject matter experts
 - project manager
- Look at header files, source code, database table definitions
- Specs and bug lists for all 3rd party tools that you use
- Prototypes, and lab notes on the prototypes

Getting Information

When There Is No Specification

- Interview development staff from the last version.
- Look at customer call records from the previous version. What bugs were found in the field?
- Usability test results
- Beta test results
- Ziff-Davis SOS CD and other tech support CD's, for bugs in your product and common bugs in your niche or on your platform
- BugNet magazine / web site for common bugs
- News Groups, CompuServe Fora, etc., looking for reports of bugs in your product and other products, and for discussions of how some features are supposed (by some) to work.
- Localization guide (probably one that is published, for localizing products on your platform.)
- Get lists of compatible equipment and environments from Marketing (in theory, at least.)
- Look at compatible products, to find their failures (then look for these in your product), how they designed features that you don't understand, and how they explain their design. See listserv's, NEWS, BugNet, etc.
- Exact comparisons with products you emulate
- Content reference materials (e.g. an atlas to check your on-line geography program)

Black Box Software Testing

Paradigms:

User Testing

User Testing

Tag line

- Strive for realism
- Let's try this with real humans (for a change).

Fundamental question or goal

- Identify failures that will arise in the hands of a person, i.e. breakdowns in the overall human/machine/software system.

Paradigmatic case(s)

- Beta testing
- In-house experiments using a stratified sample of target market
- Usability testing

User Testing

Strengths

- Design issues are more credibly exposed.
- Can demonstrate that some aspects of product are incomprehensible or lead to high error rates in use.
- In-house tests can be monitored with flight recorders (capture/replay, video), debuggers, other tools.
- In-house tests can focus on areas / tasks that you think are (or should be) controversial.

Blind spots

- Coverage is not assured (serious misses from beta test, other user tests)
- Test cases can be poorly designed, trivial, unlikely to detect subtle errors.
- Beta testing is not free, beta testers are not skilled as testers, the technical results are mixed. Distinguish marketing betas from technical betas.

Black Box Software Testing

Paradigms:

Scenario Testing

Scenario Testing

Tag lines

- “Do something useful and interesting”
- “Do one thing after another.”

Fundamental question or goal

- Challenging cases that reflect real use.

Paradigmatic case(s)

- Appraise product against business rules, customer data, competitors’ output
- Life history testing (Hans Buwalda’s “soap opera testing.”)
- Use cases are a simpler form, often derived from product capabilities and user model rather than from naturalistic observation of systems of this kind.

Scenario Testing

The ideal scenario has several characteristics:

- It is realistic (e.g. it comes from actual customer or competitor situations).
- There is no ambiguity about whether a test passed or failed.
- The test is complex, that is, it uses several features and functions.
- There is a stakeholder who has influence and will protest if the program doesn't pass this scenario.

Strengths

- Complex, realistic events. Can handle (help with) situations that are too complex to model.
- Exposes failures that occur (develop) over time

Blind spots

- Single function failures can make this test inefficient.
- Must think carefully to achieve good coverage.

Scenarios

Some ways to trigger thinking about scenarios:

- **Benefits-driven:** People want to achieve X. How will they do it, for the following X's?
- **Sequence-driven:** People (or the system) typically does task X in an order. What are the most common orders (sequences) of subtasks in achieving X?
- **Transaction-driven:** We are trying to complete a specific transaction, such as opening a bank account or sending a message. What are all the steps, data items, outputs and displays, etc.?
- **Get use ideas from competing product:** Their docs, advertisements, help, etc., all suggest best or most interesting uses of their products. How would our product do these things?

Scenarios

Some ways to trigger thinking about scenarios:

- **Competitor's output driven:** Hey, look at these cool documents they can make. Look (think of Netscape's superb handling of often screwy HTML code) at how well they display things. How do we do with these?
- **Customer's forms driven:** Here are the forms the customer produces in her business. How can we work with (read, fill out, display, verify, whatever) them?

Soap Operas

- Build a scenario based on real-life experience. This means client/customer experience.
- Exaggerate each aspect of it:
 - » example, for each variable, substitute a more extreme value
 - » example, if a scenario can include a repeating element, repeat it lots of times
 - » make the environment less hospitable to the case (increase or decrease memory, printer resolution, video resolution, etc.)
- Create a real-life story that combines all of the elements into a test case narrative.

(Thanks to Hans Buwalda for developing this approach and patiently explaining it to me.)

Soap Operas

(As these have evolved, Hans distinguishes between *normal soap operas*, which combine many issues based on user requirements—typically derived from meetings with the user community and probably don't exaggerate beyond normal use—and *killer soap operas*, which combine *and exaggerate to produce extreme cases.*)

Scenario Testing: Interesting Papers

- **Hans Buwalda on Soap Operas (in the conference proceedings of STAR East 2000)**
- **Kaner, A pattern for scenario testing, at www.testing.com**
- **Lots of literature on use cases**

Black Box Software Testing

Paradigms:

Risk-Based Testing and Risk-Based Test Management

Risk-Based Testing

Tag line

- “Find big bugs first.”

Fundamental question or goal

- Define and refine tests in terms of the kind of problem (or risk) that you are trying to manage
- Prioritize the testing effort in terms of the relative risk of different areas or issues we could test for.

Paradigmatic case(s)

- Equivalence class analysis, reformulated.
- Test in order of frequency of use.
- Stress tests, error handling tests, security tests, tests looking for predicted or feared errors.
- Sample from predicted-bugs list.
- Failure Mode and Effects Analysis (FMEA)

Equivalence and Risk

Our working definition of equivalence:

Two test cases are equivalent if you expect the same result from each.

This is fundamentally subjective. It depends on what you expect. And what you expect depends on what errors you can anticipate:

Two test cases can only be equivalent by reference to a specifiable risk.

Two different testers will have different theories about how programs can fail, and therefore they will come up with different classes.

A boundary case in this system is a “best representative.”

A best representative of an equivalence class is a test that is at least as likely to expose a fault as every other member of the class.

Risk-Based Testing

Strengths

- Optimal prioritization (assuming we correctly identify and prioritize the risks)
- High power tests

Blind spots

- Risks that were not identified or that are surprisingly more likely.
- Some “risk-driven” testers seem to operate too subjectively. How will I know what level of coverage that I’ve reached? How do I know that I haven’t missed something critical?

Evaluating Risk

- **Several approaches that call themselves “risk-based testing” ask which tests we should run and which we should skip if we run out of time.**
- **I think this is only half of the risk story. The other half focuses on test design.**
 - A key purpose of testing is to find defects. So, a key strategy for testing should be defect-based. Every test should be questioned:
 - » How will this test find a defect?
 - » What kind of defect do you have in mind?
 - » What power does this test have against that kind of defect? Is there a more powerful test? A more powerful suite of tests?

Risk-Based Testing

Many of us who think about testing in terms of risk, analogize testing of software to the testing of theories:

- Karl Popper, in his famous essay *Conjectures and Refutations*, lays out the proposition that a scientific theory gains credibility by being subjected to (and passing) harsh tests that are intended to refute the theory.
- We can gain confidence in a program by testing it harshly (if it passes the tests).
- Subjecting a program to easy tests doesn't tell us much about what will happen to the program in the field.

In risk-based testing, we create harsh tests for vulnerable areas of the program.

Risk-Based Testing

Two key dimensions:

- **Find errors** (risk-based approach to the technical tasks of testing)
- **Manage the process of finding errors** (risk-based test management)

Let's start with risk-based testing and proceed later to risk-based test management.

Risk-Based Testing: Definitions

- Hazard:
 - » A dangerous condition (something that could trigger an accident)
- Risk:
 - » Possibility of suffering loss or harm (probability of an accident caused by a given hazard).
- Accident:
 - » A hazard is encountered, resulting in loss or harm.

Risks: Where to look for errors

Quality Categories:

- Capability
- Reliability
- Usability
- Performance
- Installability
- Compatibility
- Supportability
- Testability

Each quality category is a risk category, as in: “the risk of unreliability.”

- Efficiency
- Maintainability
- Localizability
- Extendibility
- Portability

Derived from James Bach's Satisfice Model

Risks: Where to look for errors

New things: **newer features may fail.**

New technology: **new concepts lead to new mistakes.**

Learning Curve: **mistakes due to ignorance.**

Changed things: **changes may break old code.**

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.**

Tired programmers: **long overtime over several weeks or months yields inefficiencies and errors**

Adapted from James Bach's lecture notes

Risks: Where to look for errors

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

Just slipping it in: **pet feature not on plan may interact badly with other code.**

N.I.H.: **external components can cause problems.**

N.I.B.: **(not in budget) Unbudgeted tasks may be done shoddily.**

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

Adapted from James Bach's lecture notes

Risks: Where to look for errors

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

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 contract but fail product. (check out <http://www.agilealliance.org/>)

Complexity: complex code may be buggy.

Bugginess: features with many known bugs may also have many unknown bugs.

Adapted from James Bach's lecture notes

Risks: Where to look for errors

Dependencies: **failures may trigger other failures.**

Untestability: **risk of slow, inefficient testing.**

Little unit testing: **programmers find and fix most of their own bugs. Shortcutting here is a risk.**

Little system testing so far: **untested software may fail.**

Previous reliance on narrow testing strategies: **(e.g. regression, function tests), can yield a backlog of errors surviving across versions.**

Weak testing 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.**

Adapted from James Bach's lecture notes

Risks: Where to look for errors

Unfixability: **risk of not being able to fix a bug.**

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

- Bruce Webster, *Pitfalls of Object-Oriented Development*
- Michael Daconta et al. *Java Pitfalls*

Criticality: **severity of failure of very important features.**

Popularity: **likelihood or consequence if much used features fail.**

Market: **severity of failure of key differentiating features.**

Bad publicity: **a bug may appear in PC Week.**

Liability: **being sued.**

Adapted from James Bach's lecture notes

Bug Patterns as a Source of Risk

Testing Computer Software lays out a set of 480 common defects. You can use these or develop your own list.

- *Find a defect in the list*
- *Ask whether the software under test could have this defect*
- *If it is theoretically possible that the program could have the defect, 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.*

Build Your Own Model of Bug Patterns

Too many people start and end with the TCS bug list. It is outdated. It was outdated the day it was published. And it doesn't cover the issues in *your* system. Building a bug list is an ongoing process that constantly pays for itself. Here's an example from Hung Nguyen:

- This problem came up in a client/server system. The system sends the client a list of names, to allow verification that a name the client enters is not new.
- Client 1 and 2 both want to enter a name and client 1 and 2 both use the same new name. Both instances of the name are new relative to their local compare list and therefore, they are accepted, and we now have two instances of the same name.
- As we see these, we develop a library of issues. The discovery method is exploratory, requires sophistication with the underlying technology.
- Capture winning themes for testing in charts or in scripts-on-their-way to being automated.

Building Bug Patterns

There are plenty of sources to check for common failures in the common platforms

- www.bugnet.com
- www.cnet.com
- links from www.winfiles.com
- various mailing lists

Risk-Based Testing

Tasks

- Identify risk factors (hazards: ways in which the program could go wrong)
- For each risk factor, create tests that have power against it.
- Assess coverage of the testing effort program, given a set of risk-based tests. Find holes in the testing effort.
- Build lists of bug histories, configuration problems, tech support requests and obvious customer confusions.
- Evaluate a series of tests to determine what risk they are testing for and whether more powerful variants can be created.

Risk-Based Test Management

Project risk management involves

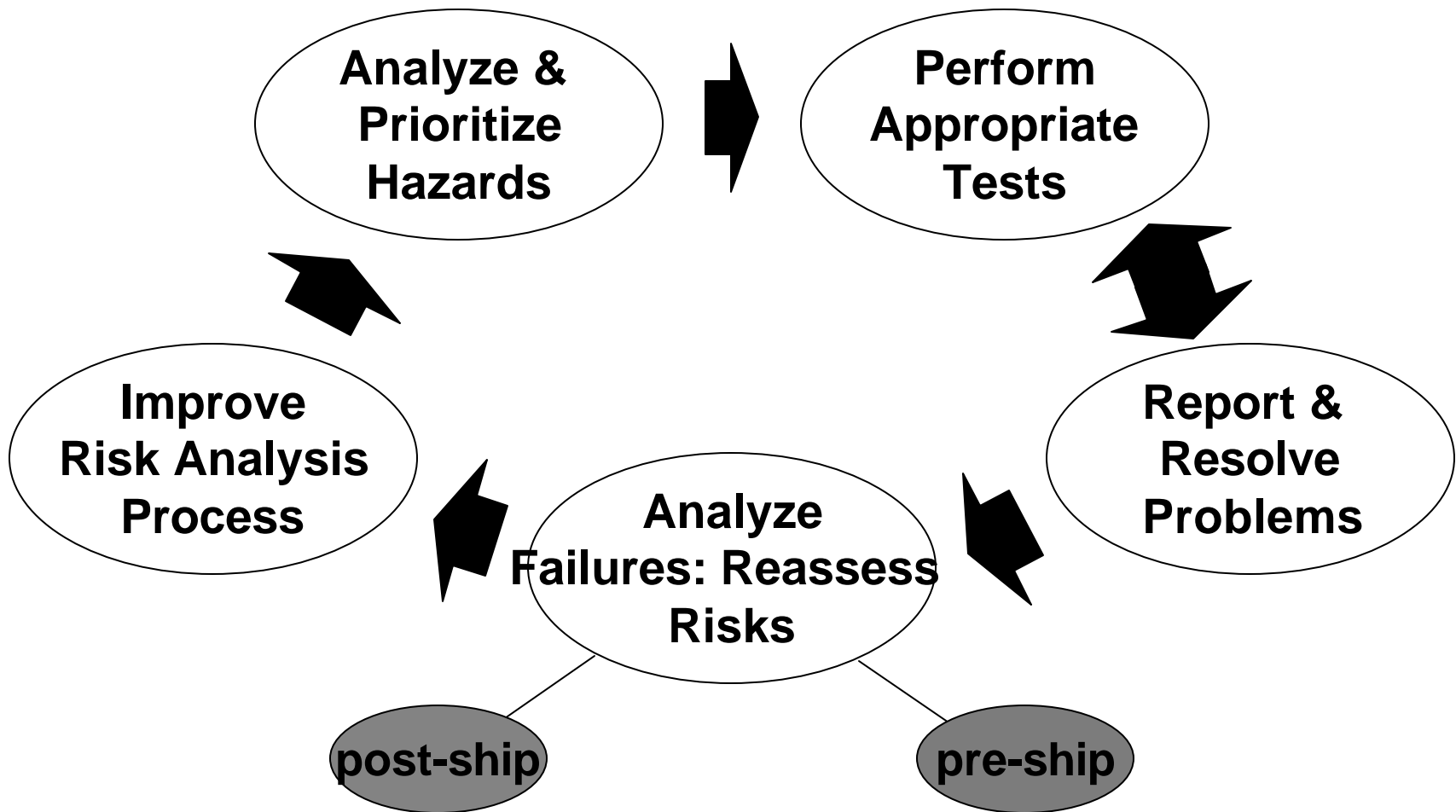
- Identification of the different risks to the project (issues that might cause the project to fail or to fall behind schedule or to cost too much or to dissatisfy customers or other stakeholders)
- Analysis of the potential costs associated with each risk
- Development of plans and actions to reduce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

Good paper by Stale Amland, *Risk Based Testing and Metrics*, 16th International Conference on Testing Computer Software, 1999.

Risk-Driven Testing Cycle





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

Risk-Based Test Management

Tasks

- List all areas of the program that could require testing
- On a scale of 1-5, assign a probability-of-failure estimate to each
- On a scale of 1-5, assign a severity-of-failure estimate to each
- For each area, identify the specific ways that the program might fail and assign probability-of-failure and severity-of-failure estimates for those
- Prioritize based on estimated risk
- Develop a stop-loss strategy for testing untested or lightly-tested areas, to check whether there is easy-to-find evidence that the areas estimated as low risk are not actually low risk.

Risk-Based Testing: Some Papers of Interest

- Stale Amland, *Risk Based Testing*
- 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*
- Carl Popper, *Conjectures & Refutations*

Black Box Software Testing

Paradigms:

Stress Testing

Stress Testing

Tag line

- “Overwhelm the product.”

Fundamental question or goal

- Learn about the capabilities and weaknesses of the product by driving it through failure and beyond. What does failure at extremes tell us about changes needed in the program’s handling of normal cases?

Paradigmatic case(s)

- Buffer overflow bugs
- High volumes of data, device connections, long transaction chains
- Low memory conditions, device failures, viruses, other crises.

Strengths

- Expose weaknesses that will arise in the field.
- Expose security risks.

Blind spots

- Weaknesses that are not made more visible by stress.

Stress Testing: Some Papers of Interest

- Astroman66, *Finding and Exploiting Bugs* 2600
- Bruce Schneier, *Crypto-Gram*, May 15, 2000
- James A. Whittaker and Alan Jorgensen, *Why Software Fails*
- James A. Whittaker and Alan Jorgensen, *How to Break Software*

Black Box Software Testing

Paradigms:

High Volume

Stochastic or Random Testing

Random / Statistical Testing

Tag line

- “High-volume testing with new cases all the time.”

Fundamental question or goal

- Have the computer create, execute, and evaluate huge numbers of tests.
 - » The individual tests are not all that powerful, nor all that compelling.
 - » Data is varied for each step.
 - » The power of the approach lies in the large number of tests.
 - » These broaden the sample, and they may test the program over a long period of time, giving us insight into longer term issues.

Random / Statistical Testing

Paradigmatic case(s)

- This is a tentative classification:
 - » NON-STOCHASTIC RANDOM TESTS
 - » STATISTICAL RELIABILITY ESTIMATION
 - » STOCHASTIC TESTS (NO MODEL)
 - » STOCHASTIC TESTS USING ON A MODEL OF THE SOFTWARE UNDER TEST
 - » STOCHASTIC TESTS USING OTHER ATTRIBUTES OF SOFTWARE UNDER TEST

Random / Statistical Testing: Non-Stochastic

Fundamental question or goal

- The computer runs a large set of essentially independent tests. The focus is on the results of each test. Tests are often designed to minimize sequential interaction among tests.

Paradigmatic case(s)

- Function equivalence testing: Compare two functions (e.g. math functions), using the second as an oracle for the first. Attempt to demonstrate that they are not equivalent, i.e. that they achieve different results from the same set of inputs.
- Other tests using fully deterministic oracles
- Other tests using heuristic oracles

Random / Statistical Testing: Statistical Reliability Estimation

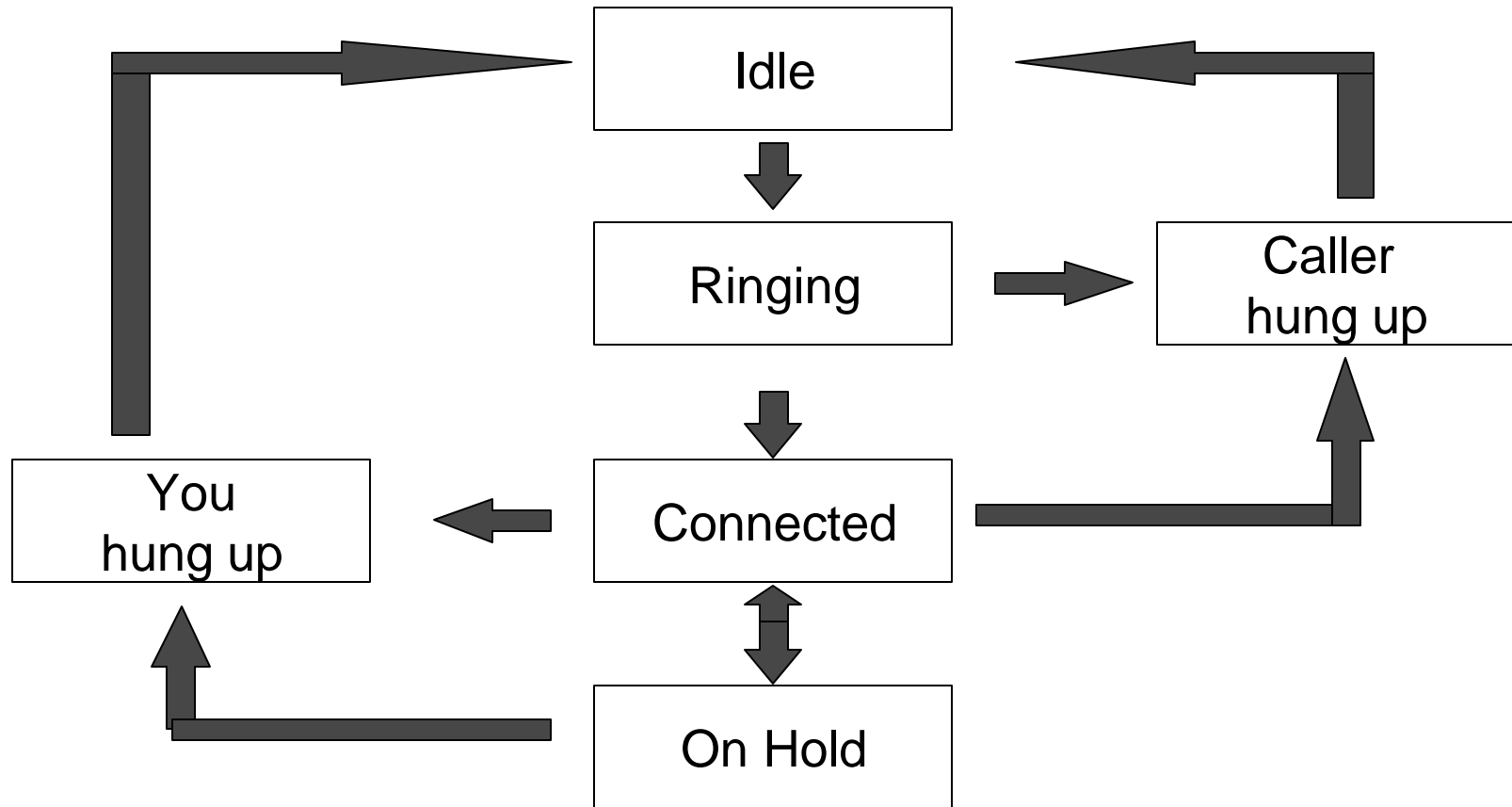
Fundamental question or goal

- Use random testing (possibly stochastic, possibly oracle-based) to estimate the stability or reliability of the software. Testing is being used primarily to qualify the software, rather than to find defects.

Paradigmatic case(s)

- Clean-room based approaches

The Need for Stochastic Testing: An Example



Random Testing: Stochastic Tests-- No Model: “Dumb Monkeys”

Fundamental question or goal

- High volume testing, involving a long sequence of tests.
- A typical objective is to evaluate program performance over time.
- The distinguishing characteristic of this approach is that the testing software does not have a detailed model of the software under test.
- The testing software might be able to detect failures based on crash, performance lags, diagnostics, or improper interaction with other, better understood parts of the system, but it cannot detect a failure simply based on the question, “Is the program doing what it is supposed to or not?”

Random Testing: Stochastic Tests-- (No Model: “Dumb Monkeys”)

Paradigmatic case(s)

- Executive monkeys: Know nothing about the system. Push buttons randomly until the system crashes.
- Clever monkeys: More careful rules of conduct, more knowledge about the system or the environment. See Freddy.
- O/S compatibility testing: No model of the software under test, but diagnostics might be available based on the environment (the NT example)
- Early qualification testing
- Life testing
- Load testing

Notes

- Can be done at the API or command line, just as well as via UI

Random Testing: Stochastic, Assert or Diagnostics Based

Fundamental question or goal

- High volume random testing using random sequence of fresh or pre-defined tests that may or may not self-check for pass/fail. The primary method for detecting pass/fail uses assertions (diagnostics built into the program) or other (e.g. system) diagnostics.

Paradigmatic case(s)

- Telephone example (asserts)
- Embedded software example (diagnostics)

Random Testing: Stochastic, Regression-Based

Fundamental question or goal

- High volume random testing using random sequence of pre-defined tests that can self-check for pass/fail.

Paradigmatic case(s)

- Life testing
- Search for specific types of long-sequence defects.

Random Testing: Stochastic, Regression-Based

Notes

- Create a series of regression tests. Design them so that they don't reinitialize the system or force it to a standard starting state that would erase history. The tests are designed so that the automation can identify failures. Run the tests in random order over a long sequence.
- This is a low-mental-overhead alternative to model-based testing. You get pass/fail info for every test, but without having to achieve the same depth of understanding of the software. Of course, you probably have worse coverage, less awareness of your actual coverage, and less opportunity to stumble over bugs.
- Unless this is very carefully managed, there is a serious risk of non-reproduceability of failures.

Random Testing: Sandboxing the Regression Tests

- In a random sequence of standalone tests, we might want to qualify each test, T1, T2, etc, as able to run on its own. Then, when we test a sequence of these tests, we know that errors are due to interactions among them rather than merely to cumulative effects of repetition of a single test.
- Therefore, for each T_i , we run the test on its own many times in one long series, randomly switching as many other environmental or systematic variables during this random sequence as our tools allow.
- We call this the “sandbox” series— T_i is forced to play in its own sandbox until it “proves” that it can behave properly on its own. (This is an 80/20 rule operation. We do want to avoid creating a big random test series that crashes only because one test doesn’t like being run or that fails after a few runs under low memory. We want to weed out these simple causes of failure. But we don’t want to spend a fortune trying to control this risk.)

Random Testing: Sandboxing the Regression Tests

Suppose that you create a random sequence of standalone tests (that were not sandbox-tested), and these tests generate a hard-to-reproduce failure.

You can run a sandbox on each of the tests in the series, to determine whether the failure is merely due to repeated use of one of them.

Random Testing: Model-based Tests

Fundamental Question or Goal

- Build a model of the software. (The analysis will reveal several defects in itself.) Generate random events / inputs to the program. Test whether the program responds as expected.

(See Model-based Testing section below)

Random Testing: Thoughts Toward an Architecture

- We have a population of tests, which may have been sandboxed and which may carry self-check info. A test series involves a sample of these tests.
- We have a population of diagnostics, probably too many to run every time we run a test. In a given test series, we will run a subset of these.
- We have a population of possible configurations, some of which can be set by the software. In a given test series, we initialize by setting the system to a known configuration. We may reset the system to new configurations during the series (e.g. every 5th test).

Random Testing: Thoughts Toward an Architecture

We can make an execution tool that takes as input

- a list of tests (or an algorithm for creating a list),
- a list of diagnostics (initial diagnostics at start of testing, diagnostics at start of each test, diagnostics on detected error, and diagnostics at end of session),
- an initial configuration and
- a list of configuration changes on specified events.

The tool runs the tests in random order and outputs results

- to a standard-format log file that defines its own structure so that
- multiple different analysis tools can interpret the same data.

Random / Statistical Testing

Strengths

- Regression doesn't depend on same old test every time.
- Partial oracles can find errors in young code quickly and cheaply.
- Less likely to miss internal optimizations that are invisible from outside.
- Can detect failures arising out of long, complex chains that would be hard to create as planned tests.

Blind spots

- Need to be able to distinguish pass from failure. Too many people think “Not crash = not fail.”
- Executive expectations must be carefully managed.
- Also, these methods will often cover many types of risks, but will obscure the need for other tests that are not amenable to automation.

Random / Statistical Testing

Blind spots

- Testers might spend much more time analyzing the code and too little time analyzing the customer and her uses of the software.
- Potential to create an inappropriate prestige hierarchy, devaluating the skills of subject matter experts who understand the product and its defects much better than the automators.

Random Testing: Some Papers of Interest

- **Larry Apfelbaum, Model-Based Testing, Proceedings of Software Quality Week 1997 (not included in the course notes)**
- **Michael Deck and James Whittaker, Lessons learned from fifteen years of cleanroom testing. STAR '97 Proceedings**
- **Doug Hoffman, Mutating Automated Tests**
- **Alan Jorgensen, An API Testing Method**
- **Noel Nyman, GUI Application Testing with Dumb Monkeys.**
- **Harry Robinson, Finite State Model-Based Testing on a Shoestring.**
- **Harry Robinson, Graph Theory Techniques in Model-Based Testing.**

Black Box Software Testing

Paradigms:

State-Model Based Testing

Random Testing: Model-based Stochastic Tests

Fundamental Question or Goal

- Build a state model of the software. (The analysis will reveal several defects in itself.) Generate random events / inputs to the program. The program responds by moving to a new state. Test whether the program has reached the expected state.

Paradigmatic case(s)

- Walking a UI menu tree using a state transition table

Random Testing: Model-based Stochastic Tests

Alan Jorgensen, Software Design Based on Operational Modes, Ph.D. thesis, Florida Institute of Technology:

“The applicability of state machine modeling to mechanical computation dates back to the work of Mealy [Mealy, 1955] and Moore [Moore, 1956] and persists to modern software analysis techniques [Mills, et al., 1990, Rumbaugh, et al., 1999]. Introducing state design into software development process began in earnest in the late 1980’s with the advent of the cleanroom software engineering methodology [Mills, et al., 1987] and the introduction of the State Transition Diagram by Yourdon [Yourdon, 1989].

“A deterministic finite automata (DFA) is a state machine that may be used to model many characteristics of a software program. Mathematically, a DFA is the quintuple, $M = (Q, S, d, q_0, F)$ where M is the machine, Q is a finite set of states, S is a finite set of inputs commonly called the “alphabet,” d is the transition function that maps $Q \times S$ to Q , q_0 is one particular element of Q identified as the initial or starting state, and $F \subseteq Q$ is the set of final or terminating states [Sudkamp, 1988]. The DFA can be viewed as a directed graph where the nodes are the states and the labeled edges are the transitions corresponding to inputs. . . .

Random Testing: Model-based Stochastic Tests

“When taking this state model view of software, a different definition of software failure suggests itself: “The machine makes a transition to an unspecified state.” From this definition of software failure a software defect may be defined as: “Code, that for some input, causes an unspecified state transition or fails to reach a required state.”

...

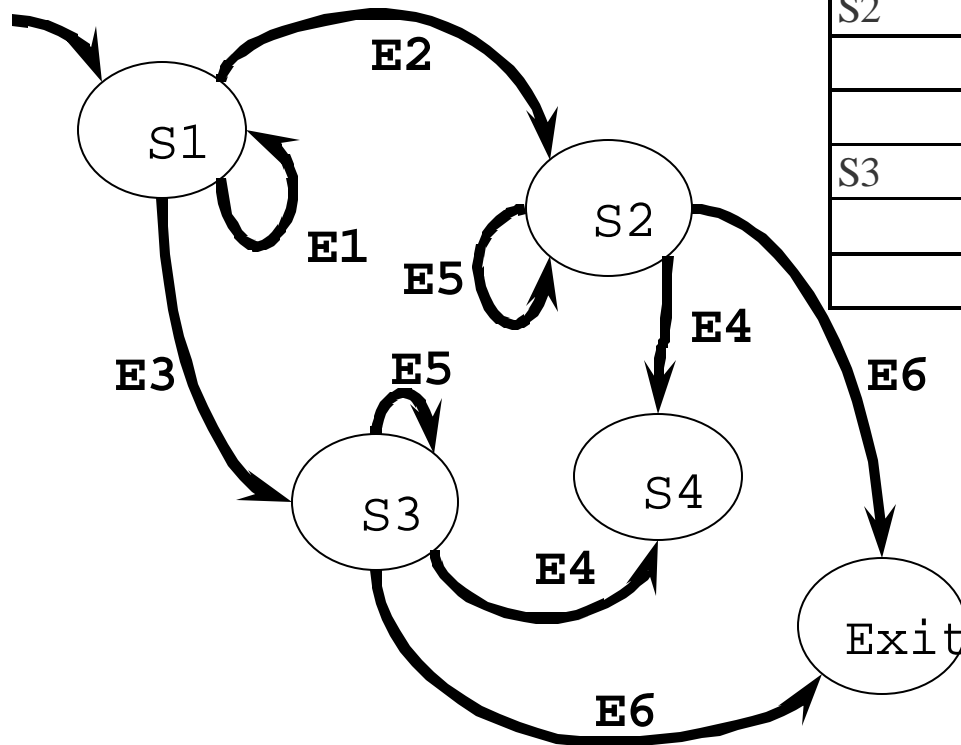
“Recent developments in software system testing exercise state transitions and detect invalid states. This work, [Whittaker, 1997b], developed the concept of an “operational mode” that functionally decomposes (abstracts) states. Operational modes provide a mechanism to encapsulate and describe state complexity. By expressing states as the cross product of operational modes and eliminating impossible states, the number of distinct states can be reduced, alleviating the state explosion problem.

Random Testing: Model-based Stochastic Tests

“Operational modes are not a new feature of software but rather a different way to view the decomposition of states. All software has operational modes but the implementation of these modes has historically been left to chance. When used for testing, operational modes have been extracted by reverse engineering.”

Alan Jorgensen, Software Design Based on Operational Modes, Ph.D. thesis, Florida Institute of Technology

State Transition Table Example



Initial State	Event	Result	New State
S1	E1	<none>	S1
	E2	logged in	S2
	E3	SU log in	S3
S2	E4	...	S4
	E5	<none>	S2
	E6	logged out	Exit
S3	E4	...	S4
	E5	admin	S3
	E6	logged out	Exit

Model Complexity

One major issue with model based testing is the complexity of the models required for most programs:

- Difficult to create the model
- Difficult to enter the model in a machine readable form
- Maintenance is a critical issue because design changes add or subtract nodes and transitions, forcing regeneration of the model.

Likely conclusions:

- Works poorly for a complex product like Word
- Likely to work well for embedded software and simple menus (think of the brakes of your car)
- In general, well suited to a limited-functionality client that will not be powered down or rebooted very often.

Model-Based Testing

Strengths

- Doesn't depend on same old test every time.
- Model unambiguously defines [part of] the product.
- Can detect failures arising out of long, complex chains that would be hard to create as planned tests.
- Tests can be reconfigured automatically by changing the model.

Blind spots

- Need to be able to distinguish pass from fail.
- Model has to match the product.
- Covers some types of risks, but can obscure the need for other tests that are not amenable to modeling or automation.

Black Box Software Testing

Paradigms: Exploratory Testing

Several of these slides are from James Bach, with permission, or from materials co-authored with James Bach

Acknowledgements

Many of the ideas in these notes were reviewed and extended by my colleagues at the 7th Los Altos Workshop on Software Testing. I appreciate the assistance of the other LAWST 7 attendees: 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.

Exploratory Testing

Tag line

- “Simultaneous learning, planning, and testing.”

Fundamental question or goal

- Software comes to tester under-documented and/or late. Tester must simultaneously learn about the product and about the test cases / strategies that will reveal the product and its defects.

Paradigmatic case(s)

- Skilled exploratory testing of the full product
- Rapid testing
- Emergency testing (including thrown-over-the-wall test-it-today testing.)
- Third party components.
- Troubleshooting / follow-up testing of defects.

Doing Exploratory Testing

- **Keep your mission clearly in mind.**
- **Distinguish between testing and observation.**
- **While testing, be aware of the limits of your ability to detect problems.**
- **Keep notes that help you report what you did, why you did it, and support your assessment of product quality.**
- **Keep track of questions and issues raised in your exploration.**

Exploratory Testing

Strengths

- Customer-focused, risk-focused
- Takes advantage of each tester's strengths
- Responsive to changing circumstances
- Well managed, it avoids duplicative analysis and testing
- High bug find rates

Blind spots

- The less we know, the more we risk missing.
- Limited by each tester's weaknesses (can mitigate this with careful management)
- This is skilled work, juniors aren't very good at it.

Problems to be Wary of...

- **Habituation may cause you to miss problems.**
- **Lack of information may impair exploration.**
- **Expensive or difficult product setup may increase the cost of exploring.**
- **Exploratory feedback loop may be too slow.**
- **Old problems may pop up again and again.**
- **High MTBF may not be achievable without well defined test cases and procedures, in addition to exploratory approach.**

Styles of Exploration

Experienced, skilled explorers develop their own styles.

When you watch or read different skilled explorers, you see very different approaches. This is a survey of the approaches that I've seen.

Styles of Exploration

- Hunches
- Models
- Examples
- Invariances
- Interference
- Error Handling
- Troubleshooting
- Group Insights
- Specifications

Styles of Exploration

- Basics
 - » **“Random”**
 - » **Questioning**
 - » **Similarity to previous errors**
 - » **Following up gossip and predictions**
 - » **Follow up recent changes**
- Models
- Examples
- Invariances
- Interference
- Error Handling
- Troubleshooting
- Group Insights
- Specifications

Random

- People who don't understand exploratory testing describe it as “random testing.” They use phrases like “random tests”, “monkey tests”, “dumb user tests”. This is probably the most common characterization of exploratory testing.
- This describes very little of the type of testing actually done by skilled exploratory testers.

Questioning

Questioning is the essence of exploration. The tester who constantly asks good questions can

- Avoid blind spots
- Quickly think of new test cases
- Constantly vary our approaches and targets
- Discover holes in specifications and product descriptions

Similarity to Previous Errors

James Bach once described exploratory testers as

mental pack rats who horde memories of every bug they've ever seen.

The way they come up with cool new tests is by analogy:

Gee, I saw a program kind of like this before, and it had a bug like this.

How could I test this program to see if it has the same old bug?

A more formal variation:

- Create a potential bugs list, like the Appendix A of *Testing Computer Software*

Another related type of analogy:

- Sample from another product's test docs.

Follow Up Gossip And Predictions

Sources of gossip:

- directly from programmers, about their own progress or about the progress / pain of their colleagues
- from attending code reviews (for example, at some reviews, the question is specifically asked in each review meeting, “What do you think is the biggest risk in this code?”)
- from other testers, writers, marketers, etc.

Sources of predictions

- notes in specs, design documents, etc. that predict problems
- predictions based on the current programmer’s history of certain types of defects

Follow Up Recent Changes

Given a current change

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

Styles of Exploration

- Hunches
- **Models**
 - » **Architecture diagrams**
 - » **Bubble diagrams**
 - » **Data relationships**
 - » **Procedural relationships**
 - » **Model-based testing (state matrix)**
 - » **Requirements definition**
 - » **Functional relationships (for regression testing)**
 - » **Failure models**
- Examples
- Invariances
- Interference
- Error Handling
- Troubleshooting
- Group Insights
- Specifications

Models and Exploration

We usually think of modeling in terms of preparation for formal testing, but there is no conflict between modeling and exploration. Both types of tests start from models. The difference is that in exploratory testing, our emphasis is on execution (try it *now*) and learning from the results of execution rather than on documentation and preparation for later execution.

Architecture Diagrams

Work from a high level design (map) of the system

- pay primary attention to interfaces between components or groups of components. We're looking for cracks that things might have slipped through
- what can we do to screw things up as we trace the flow of data or the progress of a task through the system?

You can build the map in an architectural walkthrough

- Invite several programmers and testers to a meeting. Present the programmers with use cases and have them draw a diagram showing the main components and the communication among them. For a while, the diagram will change significantly with each example. After a few hours, it will stabilize.
- Take a picture of the diagram, blow it up, laminate it, and you can use dry erase markers to sketch your current focus.
- Planning of testing from this diagram is often done jointly by several testers who understand different parts of the system.

Bubble (Reverse State) Diagrams

To troubleshoot a bug, a programmer will often work the code backwards, starting with the failure state and reading for the states that could have led to it (and the states that could have led to those).

The tester imagines a failure instead, and asks how to produce it.

- Imagine the program being in a failure state. Draw a bubble.
- What would have to have happened to get the program here? Draw a bubble for each immediate precursor and connect the bubbles to the target state.
- For each precursor bubble, what would have happened to get the program there? Draw more bubbles.
- More bubbles, etc.
- Now trace through the paths and see what you can do to force the program down one of them.

Bubble (Reverse State) Diagrams

Example:

How could we produce a paper jam (as a result of defective firmware, rather than as a result of jamming the paper?) The laser printer feeds a page of paper at a steady pace. Suppose that after feeding, the system reads a sensor to see if there is anything left in the paper path. A failure would result if something was wrong with the hardware or software controlling or interpreting the paper feeding (rollers, choice of paper origin, paper tray), paper size, clock, or sensor.

Data Relationships

- Pick a data item
- Trace its flow through the system
- What other data items does it interact with?
- What functions use it?
- Look for inconvenient values for other data items or for the functions, look for ways to interfere with the function using this data item

Data Relationship Chart

<i>Field</i>	<i>Entry Source</i>	<i>Display</i>	<i>Print</i>	<i>Related Variable</i>	<i>Relationship</i>
Variable 1	<i>Any way you can change values in V1</i>	<i>After V1 & V2 are brought to incompatible values, what are all the ways to display them?</i>	<i>After V1 & V2 are brought to incompatible values, what are all the ways to display or use them?</i>	Variable 2	Constraint to a range
Variable 2	<i>Any way you can change values in V1</i>			Variable 1	Constraint to a range

Procedural Relationships

- Pick a task
- Step by step, describe how it is done and how it is handled in the system (to as much detail as you know)
- Now look for ways to interfere with it, look for data values that will push it toward other paths, look for other tasks that will compete with this one, etc.

Improvisational Testing

The originating model here is of the test effort, not (explicitly) of the software.

Another approach to ad hoc testing is to treat it as improvisation on a theme, not unlike jazz improvisation in the musical world. For example, testers often start with a Test Design that systematically walks through all the cases to be covered. Similarly, jazz musicians often start with a musical score or “lead sheet” for the tunes on which they intend to improvise.

In this version of the ad hoc approach, the tester is encouraged to take off on tangents from the original Test Design whenever it seems worthwhile. In other words, the tester uses the test design but invents variations. This approach combines the strengths of both structured and unstructured testing: the feature is tested as specified in the test design, but several variations and tangents are also tested. On this basis, we expect that the improvisational approach will yield improved coverage.

Improvisational Testing

The originating model here is of the test effort, not (explicitly) of the software.

Improvisational techniques are also useful when verifying that defects have been fixed. Rather than simply verifying that the steps to reproduce the defect no longer result in the error, the improvisational tester can test more deeply “around” the fix, ensuring that the fix is robust in a more general sense.

**Johnson & Agruss, *Ad Hoc Software Testing:
Exploring the Controversy of Unstructured Testing*
STAR'98 WEST**

State Model-Based Testing

Notes from Harry Robinson & James Tierney

By modeling specifications, drawing finite state diagrams of what we thought was important about the specs, or just looking at the application or the API, we can find orders of magnitude more bugs than traditional tests.

Example, they spent 5 hours looking at the API list, found 3-4 bugs, then spent 2 days making a model and found 272 bugs. The point is that you can make a model that is too big to carry in your head. Modeling shows inconsistencies and illogicalities.

Look at

- all the possible inputs the software can receive, then
- all the operational modes, (something in the software that makes it work differently if you apply the same input)
- all the actions that the software can take.
- Do the cross product of those to create state diagrams so that you can see and look at the whole model.
- Use to do this with dozens and hundreds of states, Harry has a technique to do thousands of states.

[www.geocities.com/model based testing](http://www.geocities.com/model_based_testing)

Using a Model of the Requirements to Drive Test Design

Notes from Melora Svoboda

Requirements model based on Gause / Weinberg. Developing a mind map of requirements, you can find missing requirements before you see code.

Business requirements

- Issues
- Assumptions
- Choices
- <<<< the actual problem >>>>

Customer Problem Definition

- USERS (nouns)
 - *avored*
 - *disavored*
 - *ignored*
- ATTRIBUTES (adjectives)
 - » defining
 - » optimizing

Using a Model of the Requirements to Drive Test Design

Notes from Melora Svoboda

Customer Problem Definition (continued)

- FUNCTIONS (verbs)
 - » hidden
 - » evident

The goal is to test the assumptions around this stuff, and discover an inventory of hidden functions.

Comment: This looks to me (Kaner) like another strategy for developing a relatively standard series of questions that fall out of a small group of categories of analysis, much like the Satisfice model. Not everyone finds the Satisfice model intuitive. If you don't, this might be a usefully different starting point.

Functional Relationships

(More notes from Melora)

A model (what you can do to establish a strategy) for deciding how to decide what to regression test after a change:

1. Map program structure to functions.
 - This is (or would be most efficiently done as) a glass box task. Learn the internal structure of the program well enough to understand where each function (or source of functionality) fits.
2. Map functions to behavioral areas (expected behaviors)
 - The program misbehaved and a function or functions were changed. What other behaviors (visible actions or options of the program) are influenced by the functions that were changed?
3. Map impact of behaviors on the data
 - When a given program behavior is changed, how does the change influence visible data, calculations, contents of data files, program options, or anything else that is seen, heard, sent, or stored?

Failure Model: Whittaker: “The fundamental cause of software errors”

Constraint violations

- input constraints
 - » such as buffer overflows
- output constraints
- computation
 - » look for divide by zeros and rounding errors.
Figure out inputs that you give the system that will make it not recognize the wrong outputs.
- data violations
- Really good for finding security holes

Styles of Exploration

- Hunches
- Models
- **Examples**
 - » **Use cases**
 - » **Simple walkthroughs**
 - » **Positive testing**
 - » **Scenarios**
 - » **Soap operas**
- Invariances
- Interference
- Error Handling
- Troubleshooting
- Group Insights
- Specifications

Use Cases

- List the users of the system
- For each user, think through the tasks they want to do
- Create test cases to reflect their simple and complex uses of the system

Simple Walkthroughs

Test the program broadly, but not deeply.

- Walk through the program, step by step, feature by feature.
- Look at what's there.
- Feed the program simple, nonthreatening inputs.
- Watch the flow of control, the displays, etc.

Positive Testing

- Try to get the program working in the way that the programmers intended it.
- One of the points of this testing is that you educate yourself about the program. You are looking at it and learning about it from a sympathetic viewpoint, using it in a way that will show you what the value of the program is.
- This is true “positive” testing—you are trying to make the program show itself off, not just trying to confirm that all the features and functions are there and kind of sort of working.

Scenarios

The ideal scenario has several characteristics:

- It is realistic (e.g. it comes from actual customer or competitor situations).
- There is no ambiguity about whether a test passed or failed.
- The test is complex, that is, it uses several features and functions.
- There is a stakeholder who will make a fuss if the program doesn't pass this scenario.

For more on scenarios, see the scenarios paradigm discussion.

Styles of Exploration

- Hunches
- Models
- Examples
- **Invariances**
- Interference
- Error Handling
- Troubleshooting
- Group Insights
- Specifications

Invariances

These are tests run by making changes that shouldn't affect the program. Examples:

- load fonts into a printer in different orders
- set up a page by sending text to the printer and then the drawn objects or by sending the drawn objects and then the text
- use a large file, in a program that should be able to handle any size input file (and see if the program processes it in the same way)
- mathematical operations in different but equivalent orders

=====

John Musa — Intro to his book, *Reliable Software Engineering*, says that you should use different values within an equivalence class. For example, if you are testing a flight reservation system for two US cities, vary the cities. They shouldn't matter, but sometimes they do.

Styles of Exploration

- Hunches
- Models
- Examples
- Invariances
- **Interference**
 - » **Interrupt**
 - » **Change**
 - » **Stop**
 - » **Pause**
 - » **Swap**
 - » **Compete**
- Error Handling
- Troubleshooting
- Group Insights
- Specifications

Interference Testing

We're often looking 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.

Interrupt

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

Change

Change something that 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

Stop

- 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

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)
- 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.

Swap (out of memory)

- Swap the process out of memory while it is running (e.g. change focus to another application and 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.

Compete

Examples:

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

Styles of Exploration

- Hunches
- Models
- Examples
- Invariances
- Interference
- **Error Handling**
- Troubleshooting
- Group Insights
- Specifications

Error Handling

The usual suspects:

- Walk through the error list.
 - » Press the wrong keys at the error dialog.
 - » Make the error several times in a row (do the equivalent kind of probing to defect follow-up testing).
- Device-related errors (like disk full, printer not ready, etc.)
- Data-input errors (corrupt file, missing data, wrong data)
- Stress / volume (huge files, too many files, tasks, devices, fields, records, etc.)

Styles of Exploration

- Hunches
- Models
- Examples
- Invariances
- Interference
- Error Handling
- **Troubleshooting**
- Group Insights
- Specifications

Troubleshooting

We often do exploratory tests when we troubleshoot bugs:

- Bug analysis:
 - » simplify the bug by deleting or simplifying steps
 - » simplify the bug by simplifying the configuration (or the tools in the background)
 - » clarify the bug by running variations to see what the problem is
 - » clarify the bug by identifying the version that it entered the product
 - » strengthen the bug with follow-up tests (using repetition, related tests, related data, etc.) to see if the bug left a side effect
 - » strengthen the bug with tests under a harsher configuration
- Bug regression: vary the steps in the bug report when checking if the bug was fixed

Styles of Exploration

- Hunches
- Models
- Examples
- Invariances
- Interference
- Error Handling
- Troubleshooting
- **Group Insights**
 - » **Brainstormed test lists**
 - » **Group discussion of related components**
 - » **Fishbone analysis**
- Specifications

Brainstormed Test Lists

We saw a simple example of this at the start of the class. You brainstormed a list of tests for the two-variable, two-digit problem:

- The group listed a series of cases (test case, why)
- You then examined each case and the class of tests it belonged to, looking for a more powerful variation of the same test.
- You then ran these tests.

You can apply this approach productively to any part of the system.

Group Discussion of Related Components

The objective is to test the interaction of two or more parts of the system.

The people in the group are very familiar with one or more of parts. Often, no one person is familiar with all of the parts of interest, but collectively the ideal group knows all of them.

The group looks for data values, timing issues, sequence issues, competing tasks, etc. that might screw up the orderly interaction of the components under study.

Fishbone Analysis

- Fishbone analysis is a traditional failure analysis technique. Given that the system has shown a specific failure, you work backwards through precursor states (the various paths that could conceivably lead to this observed failure state).
- As you walk through, you say that Event A couldn't have happened unless Event B or Event C happened. And B couldn't have happened unless B1 or B2 happened. And B1 couldn't have happened unless X happened, etc.
- While you draw the chart, you look for ways to prove that X (whatever, a precursor state) *could* actually have been reached. If you succeed, you have found one path to the observed failure.
- As an exploratory test tool, you use “risks” instead of failures. You imagine a possible failure, then walk backwards asking if there is a way to achieve it. You do this as a group, often with a computer active so that you can try to get to the states as you go.

Styles of Exploration

- Hunches
- Models
- Examples
- Invariances
- Interference
- Error Handling
- Troubleshooting
- Group Insight
- **Specifications**
 - » **Active reading -- Satisfice Method**
 - » **Active reading -- Ambiguity analysis**
 - » **User manual**
 - » **Consistency heuristics**

Active Reading

- James Bach's satisfice testing model (details at Satisfice.com).
- You can use this method to discover faults in a specification, such as holes, ambiguities, and contradictions.
- The goal is to constantly question the spec, identifying statements about product, project and risk, but also identifying missing details and unrealistic discussions.
- Anything you flag as an issue (or write a question about), is a candidate for exploratory testing.

Active Reading

(Ambiguity Analysis)

There are all sorts of sources of ambiguity in software design and development.

- In the wording or interpretation of specifications or standards
- In the expected response of the program to invalid or unusual input
- In the behavior of undocumented features
- In the conduct and standards of regulators / auditors
- In the customers' interpretation of their needs and the needs of the users they represent
- In the definitions of compatibility among 3rd party products

Whenever there is ambiguity, there is a strong opportunity for a defect (at least in the eyes of anyone who understands the world differently from the implementation).

One interesting workbook: Cecile Spector, *Saying One Thing, Meaning Another*.

User Manual

Write part of the user manual and check the program against it as you go. Any writer will discover bugs this way. An exploratory tester will discover quite a few this way.

Consistency Heuristics:

Consistent with History: **Present function behavior is consistent with past behavior.**

Consistent with an Image: **Function behavior is consistent with an image that the organization wants to project.**

Consistent with Comparable Products: **Function behavior is consistent with that of similar functions in comparable products.**

Consistent with Claims: **Function behavior is consistent with what people say it's supposed to be.**

Consistent with User Values: **Function behavior is consistent with what we think users want.**

Consistent within Product: **Function behavior is consistent with behavior of comparable functions or functional patterns within the product.**

Consistent with Purpose: **Function behavior is consistent with its apparent purpose.**

Exploratory Testing: Some Papers of Interest

- Chris Agruss & Bob Johnson, *Ad Hoc Software Testing Exploring the Controversy of Unstructured Testing*
- Cem Kaner & James Bach, *Exploratory Testing. (Available later in the materials)*
- Whittaker, *How to Break Software*

Notes
