Rapid Test Planning

Cem Kaner, J.D., Ph.D.
Department of Computer Sciences
Florida Institute of Technology

James Bach
Satisfice, Inc.

October, 2001
STAR West

Acknowledgments

- These notes outline the test planning chapters in prep for ***Testing Computer Software***, 3rd Ed., by Cem Kaner, James Bach, Hung Quoc Nguyen, Jack Falk, Brian Lawrence & Bob Johnson. They incorporate and adapt materials by these authors. The notes are also based on materials developed for ***Lessons Learned in Software Testing***, a book just completed by Cem Kaner, James Bach and Bret Pettichord.
- Many of the ideas in these notes were reviewed and refined at the Third Los Altos Workshop on Software Testing (LAWST), February 7-8, 1998, and at the Eleventh LAWST, October 28-29, 2000.
  - The participants at LAWST 3 were: Chris Agruss, James Bach, Karla Fisher, David Gelperin, Kenneth Groder, Elisabeth Hendrickson, Doug Hoffman, III (recorder), Bob Johnson, Cem Kaner (host), Brian Lawrence (facilitator), Brian Marick, Thanga Meenakshi, Noel Nyman, Jeffery E. Payne, Bret Pettichord, Johanna Rothman, Jane Stepak, Melora Svoboda, Jeremy White, and Rodney Wilson.
  - The participants at LAWST 11 were: Chris Agruss, James Bach, Hans Buwalda, Marge Farrell. Sam Guckenheimer, Elisabeth Hendrickson, Doug Hoffman, III (recorder), Bob Johnson, Karen Johnson, Cem Kaner (host), Brian Lawrence (facilitator), Alan Myrvold, Hung Quoc Nguyen, Noel Nyman, Neal Reizer, Amit Singh, and Melora Svoboda.

Abstract

This workshop has grown out of our dissatisfaction with paper-intensive approaches that attempt to provide a seemingly reproducible, somewhat mechanical process for planning and managing testing and test documentation. Over the past 17 years, we have criticized IEEE standard 829 (on software test documentation) and related approaches as being often inappropriate.

Colleagues have asked what we would put in IEEE 829's place. To date, our responses have been piecemeal. This seminar's notes are a draft of our attempt to write a more comprehensive response.

- They start from the premise that the best approach to test documentation depends on the project context. For example, creating detailed test documentation can be useful for some projects but can get in the way of the development of a high-volume automated testing strategy. *What are the relevant differences between these projects?* Before adopting an implementation guideline (like IEEE 829), we should analyze our requirements. There is no point spending a fortune on creating a deliverable (here, the test documentation set) that will not be used or that will interfere with the efficient running of the project. Instead, we should build a documentation set that will actually satisfy the real needs of the project.
- The notes also reflect our view that testing is an exercise in critical thinking and careful questioning. A test case is a question that you ask of the program *(Are you broken in this way?).* The point of a test case is to reduce uncertainty associated with the product. (A test is good if it will reduce uncertainty, whether it finds a bug or not.) A test plan is a structure for asking questions of the project and the product. These notes suggest strategies for asking better questions, and they provide useful clusters of questions.
- The notes also provide samples of some common test planning documents, such as tables and matrices. These will probably be among the building blocks of any testing program that you set up.

What Makes Our Approach "Rapid"?

- Focus on your actual requirements not requirements dictated by standards
- Focus on gathering and using information rather than writing down all possible details
- Iterative development of materials, adding detail and depth to test cases as you go

Overview

- Problems with the (allegedly) standard approach
- Defining your documentation requirements
- A model for testing and test documentation
- Test documentation elements

Problems with the (allegedly) standard approach

- IEEE Standard 829 for Software Test Documentation
  - Test plan
  - Test-design specification
  - ***Test-case specification***
    - ***Test-case specification identifier***
    - ***Test items***
    - ***Input specifications***
    - ***Output specifications***
    - ***Environmental needs***
    - ***Special procedural requirements***
    - ***Intercase dependencies***
  - Test-procedure specification
  - Test-item transmittal report
  - Test-log

We often see one or more pages per test case.

Problems with the (allegedly) standard approach

- What is the documentation cost per test case?
- What is the maintenance cost of the documentation, per test case?
- If software design changes create documentation maintenance costs, how much inertia do we build into our system? How much does extensive test documentation add to the cost of late improvement of the software? How much should we add?
- What inertia is created in favor of invariant regression testing?
- Is this incompatible with exploratory testing? Do we always want to discourage exploration?

Problems with the (allegedly) standard approach

- What is the impact on high-volume test automation?
- How often do project teams start to follow 829 but then give it up mid-project? What does this do to the net quality of the test documentation and test planning effort?
- WHAT REQUIREMENTS DOES A STANDARD LIKE THIS FULFILL?
- WHICH STAKEHOLDERS GAIN A NET BENEFIT FROM IEEE STANDARD DOCUMENTATION?
- WHAT BENEFITS DO THEY GAIN, AND WHY ARE THOSE BENEFITS IMPORTANT TO THEM?

Problems with the (allegedly) standard approach

It is essential to understand your requirements for test documentation.

Unless following a "standard" helps you meet your requirements, it is empty at best, anti-productive at worst.

Requirements

- There are many different notions of what a good set of test documentation would include. Before spending a substantial amount of time and resources, it's worth asking what documentation should be developed (and why?)
- *Test documentation is expensive and it takes a long time to produce. If you figure out some of your main requirements first, you might be able to do your work in a way that achieves them.*

Defining documentation requirements

- Stakeholders, interests, actions, objects
  - Who would use or be affected by test documentation?
  - What interests of theirs does documentation serve or disserve?
  - What will they do with the documentation?
  - What types of documents are of high or low value?
- Asking questions
- Context-free questions
- Context-free questions specific to test planning
- Evaluating a plan

Discovering Requirements

- Requirements
  - Anything that drives or constrains design
- Stakeholders
  - Favored, disfavored, and neutral stakeholders
- Stakeholders' interests
  - Favored, disfavored, and neutral interests
- Actions
  - Actions support or interfere with interests
- Objects

Exercise

- 1. List the Stakeholders
  - Favored
  - Disfavored
  - Neutral stakeholders
- 2. For each Stakeholder, list her Interests
  - Favored
  - Disfavored
  - Neutral interests
- 3. For each Interest, list Actions
  - Actions support an interest
  - Actions interfere with an interest

Exercise

- Objects: The Stuff You Create
  - Such as features, data of the product
- For each object, what is its relationship
  - to a stakeholder,
  - a stakeholder's interest, or
  - in the actions the stakeholder wants to take or will have taken on her?

Testers' Questions: Does Your Car Work?

HOW CAN YOU TELL THAT SOMETHING WORKS?

- How do you know your car works?
- Are there situations in which your car would stop working?
- Who else uses your car? Do they use it differently than you, so that it might work for you but fail for them?
- What facts would cause you to believe that your car *doesn't* work?
- In what ways could your car *not* work, yet seem to you that it does?
- In what ways could your car work, yet seem to you that it *doesn't*?
- Do you know enough about cars to answer these questions?
- Have you observed your car enough, *today*, to answer them?
- Under what circumstances would these questions matter?

Questioning

- Requirements analysis requires information gathering
  - Read books on consulting
  - Gause & Weinberg, *Exploring Requirements* is an essential source on context-free questioning
- There are many types of questions:
  - Open vs. closed
  - Hypothetical vs. behavioral
  - Opinion vs. factual
  - Historical vs. predictive
  - Context-dependent and context-free

The classic context-free questions

- The traditional newspaper reporters' questions are:
  - Who
  - What
  - When
  - Where
  - How
  - Why
- For example, *Who will use this feature? What does this user want to do with it? Who else will use it? Why? Who will choose not to use it? What do they lose? What else does this user want to do in conjunction with this feature? Who is not allowed to use this product or feature, why, and what security is in place to prevent them?*
- We use these in conjunction with questions that come out of the testing model (see below). The model gives us a starting place. We expand it by asking each of these questions as a follow-up to the initial question.

Context-Free Questions: Defining the Problem

Based on: *The CIA's Phoenix Checklists (Thinkertoys, p. 140) and Bach's Evaluation Strategies (Rapid Testing Course notes)*

- Why is it necessary to solve the problem?
- What benefits will you receive by solving the problem?
- What is the unknown?
- What is it that you don't yet understand?
- What is the information that you have?
- What is the source of this problem? (Specs? Field experience? An individual stakeholder's preference?)
- Who are the stakeholders?
- How does it relate to which stakeholders?
- What isn't the problem?
- Is the information sufficient? Or is it insufficient? Or redundant? Or contradictory?
- Should you draw a diagram of the problem? A figure?

Context-Free Questions: Defining the Problem

- Where are the boundaries of the problem?
- What product elements does it apply to?
- How does this problem relate to the quality criteria?
- Can you separate the various parts of the problem? Can you write them down? What are the relationships of the parts of the problem?
- What are the constants (things that can't be changed) of the problem?
- What are your critical assumptions about this problem?
- Have you seen this problem before?
- Have you seen this problem in a slightly different form?
- Do you know a related problem?
- Try to think of a familiar problem having the same or a similar unknown.
- Suppose you find a problem related to yours that has already been solved. Can you use it? Can you use its method?
- Can you restate your problem? How many different ways can you restate it? More general? More specific? Can the rules be changed?
- What are the best, worst, and most probable cases you can imagine?

Context-Free Questions

Context-free process questions

- Who is the client?
- What is a successful solution worth to this client?
- What is the real (underlying) reason for wanting to solve this problem?
- Who can help solve the problem?
- How much time is available to solve the problem?

Context-free product questions

- What problems could this product create?
- What kind of precision is required / desired for this product?

Metaquestions (when interviewing someone for info)

- Am I asking too many questions?
- Do my questions seem relevant?
- Are you the right person to answer these questions?
- Is there anyone else who can provide additional information?
- Is there anything else I should be asking?
- Is there anything you want to ask me?
- May I return to you with more questions later?

***A sample of additional questions based on Gause & Weinberg's* Exploring Requirements p. 59-64**

What is your group's mission?

- Find important problems
- Assess quality
- Certify to standard
- Fulfill process mandates
- Satisfy stakeholders
- Assure accountability
- Advise about QA
- Advise about testing
- Advise about quality
- Maximize efficiency
- Minimize time
- Minimize cost

The quality of testing depends on which of these possible missions matter and how they relate.

Many debates about the goodness of testing are really debates over missions and givens.

Test Docs Requirements Questions

- **Is test documentation a product or tool?**
- **Is software quality driven by legal issues or by market forces?**
- **How quickly is the design changing?**
- **How quickly does the specification change to reflect design change?**
- **Is testing approach oriented toward proving conformance to specs or nonconformance with customer expectations?**
- **Does your testing style rely more on already-defined tests or on exploration?**
- **Should test docs focus on what to test (objectives) or on how to test for it (procedures)?**
- **Should the docs ever control the testing project?**

Test Docs Requirements Questions

- **If the docs control parts of the testing project, should that control come early or late in the project?**
- **Who are the primary readers of these test documents and how important are they?**
- **How much traceability do you need? What docs are you tracing back to and who controls them?**
- **To what extent should test docs support tracking and reporting of project status and testing progress?**
- **How well should docs support delegation of work to new testers?**
- **What are your assumptions about the skills and knowledge of new testers?**
- **Is test doc set a process model, a product model, or a defect finder?**

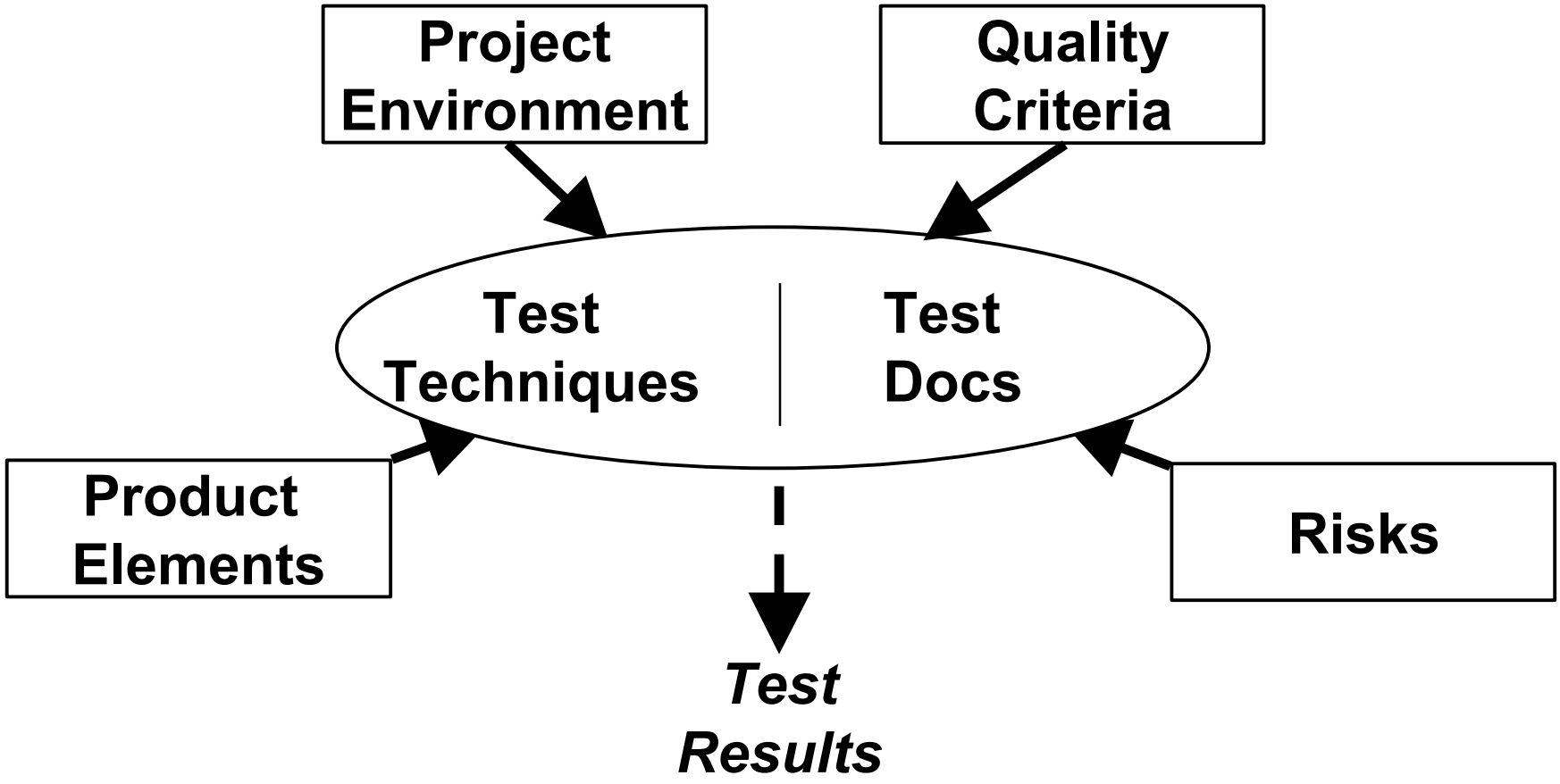
Test Docs Requirements Questions

- **A test suite should provide prevention, detection, and prediction. Which is the most important for this project?**
- **How maintainable are the test docs (and their test cases)? And, how well do they ensure that test changes will follow code changes?**
- **Will the test docs help us identify (and revise/restructure in face of) a permanent shift in the risk profile of the program?**
- **Are (should) docs (be) automatically created as a byproduct of the test automation code?**

Ultimately, write a mission statement

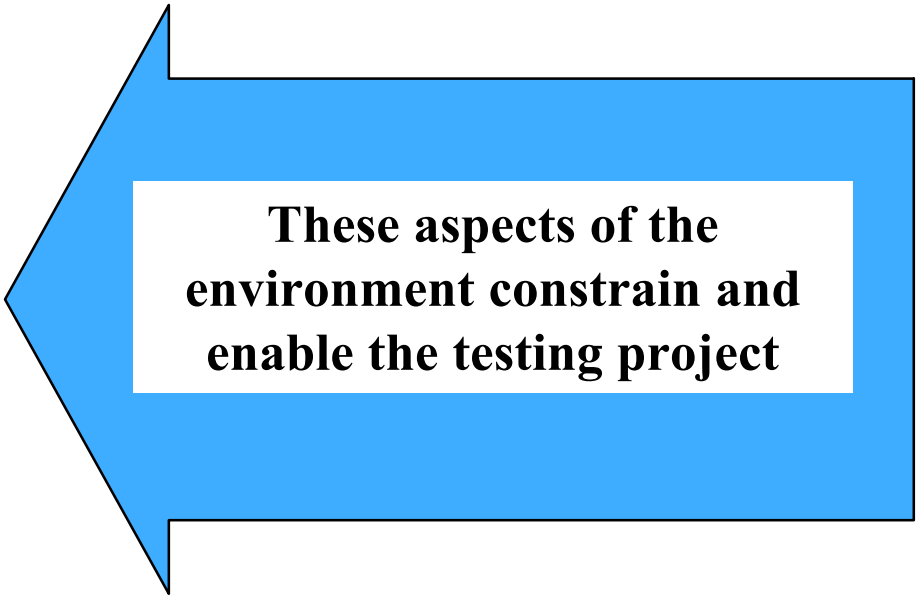
- Try to describe your core documentation requirements in one sentence that doesn't have more than three components.
- Examples:
  - The test documentation set will primarily support our efforts to find bugs in this version, to delegate work, and to track status.
  - The test documentation set will support ongoing product and test maintenance over at least 10 years, will provide training material for new group members, and will create archives suitable for regulatory or litigation use.

A Model of Software Testing



Project Environment Factors:

- Stakeholders
- Processes
- Staff
- Schedules
- Equipment
- Tools & Test Materials
- Information
- Items Under Test
- Logistics
- Budget
- Deliverables



Project Factors

- **Stakeholders:**
  - ***Anyone who is a client of the main project***
  - ***Anyone who is a client of the testing project***
    - Includes customers (purchasers), end users, tech support, programmers, project mgr, doc group, etc.
- **Processes:**
  - ***The tasks and events that comprise the main project***
    - How the overall project is run
  - ***The tasks and events that comprise the test project***
    - How the testing project is run
- **Staff:**
  - ***Everyone who helps develop the product***
    - *Sources of information and assistance*
  - ***Everyone who will perform or support testing***
    - *Special talents or experiences of team members*
    - *Size of the group*
    - *Extent to which they are focused or are multi-tasking*
    - *Organization: collaboration & coordination of the staff*
    - *Is there an independent test lab?*

Project Factors

- **Schedules:** ***The sequence, duration and synchronization of events***
  - When will testing start and how long is it expected to take?
  - When will specific product elements be available to test?
  - When will devices or tools be available to support testing?
- **Equipment:** ***Hardware required for testing***
  - What devices do we need to test the product with? Do we have them?
- **Tools & Test Materials:** ***Software required or desired for testing.***
  - Automation: Are such tools available? Do we want to use them? Do we have them? Do we understand them?
  - Probes or diagnostics to help observe the product under test?
  - Matrices, checklists, other testing documentation?
- **Information:** ***(As needed for testing) about the project or product.***
  - Specifications, requirements documents, other reference materials to help us determine pass/fail or to credibly challenge odd behaviour.
    - What is the availability of these documents?
    - What is the volatility of these documents?

Project Factors

- **Items Under Test:** ***Anything that will be tested***
  - For each product element:
    - Is it available (or when will it be)?
    - Is it volatile (and what is the change process)?
    - Is it testable?
- **Logistics:** ***Facilities and support needed for organizing and conducting the testing***
  - Do we have the supplies / physical space, power, light / security systems (if needed) / procedures for getting more?
- **Budget:** ***Money and other resources for testing***
  - Can we afford the staff, space, training, tools, supplies, etc.?
- **Deliverables:** ***The observable products of the test project***
  - Such as bug reports, summary reports, test documentation, master disk.
    - What are you supposed to create and can you do it?
  - Will we archive the items under test and other products of testing?

Product Elements: **A product is…**

An experience or solution provided to a customer.

Everything that comes in the box, plus the box!

Functions and data, executed on a platform, that serve a purpose for a user.

1 A software product is much more than code.
2 It involves a purpose, platform, and user.
3 It consists of many interdependent *elements*.

Product Elements:

- **Structures:** ***Everything that comprises the physical product***
  - Code: the code structures that comprise the product, from executables to individual routines
  - Interfaces: points of connection and communication between subsystems
  - Hardware: hardware components integral to the product
  - Non-executable files: any files other than programs, such as text files, sample data, help files, etc.
  - Alternate Media: anything beyond software and hardware, such as paper documents, web links and content, packaging, license agreements, etc.

Product Elements:

- **Functions: *Everything that the product does*.**
  - User Interface: functions that mediate the exchange of data with the user
  - System Interface: functions that exchange data with something other than the user, such as with other programs, hard disk, network, printer, etc.
  - Application: functions that define or distinguish the product or fulfill core requirements
  - Error Handling: functions that detect and recover from errors, including error messages
  - Testability: functions provided to help test the product, such as diagnostics, log files, asserts, test menus, etc.
- **Temporal relationships: *How the program functions over time***
  - Sequential operation: state-to-state transitions
  - Data: changes in variables over time
  - System interactions: such as synchronization or ordering of events in distributed systems

Product Elements:

- **Data: *Everything that the product processes***
  - Input: data that is processed by the product
  - Output: data that results from processing by the product
  - Preset: data supplied as part of the product or otherwise built into it, such as prefab databases, default values, etc.
  - Persistent: data stored internally and expected to persist over multiple operations. This includes modes or states of the product, such as options settings, view modes, contents of documents, etc.
  - Temporal: data based on time, such as date stamps or number of events recorded in a unit of time

Product Elements:

- **Platform:** ***Everything on which the product depends***
  - External Hardware: components and configurations that are not part of the shipping product, but are required (or optional) in order for the product to work. Includes CPU's, memory, keyboards, peripheral boards, etc.
  - External Software: software components and configurations that are not a part of the shipping product, but are required (or optional) in order for the product to work. Includes operating systems, concurrently executing applications, drivers, fonts, etc.
- **Operations:** ***How the product will be used***
  - Usage Profile: the pattern of usage, over time, including patterns of data that the product will typically process in the field. This varies by user and type of user.
  - Environment: the physical environment in which the product will be operated, including such elements as light, noise, and distractions.

Product Elements: Coverage

Product coverage is the proportion of the product that has been tested.

- **There are as many kinds of coverage as there are ways to model the product.**
  - Structural
  - Functional
  - Temporal
  - Data
  - Platform
  - Operations

See Software Negligence & Testing Coverage at www.kaner.com for 101 examples of coverage "measures."

Quality Criteria

- Capability
- Reliability
- Usability
- Performance
- Installability
- Compatibility
- Supportability
- Testability
- Maintainability
- Portability
- Localizability
- Efficiency

Quality is value to some person -- Jerry Weinberg

Quality Criteria

- Accessibility
- Capability
- Compatibility
- Concurrency
- Conformance to Standards
- Efficiency
- Installability and uninstallability
- Localizability
- Maintainability
- Performance
- Portability
- Recoverability
- Reliability
- Scalability
- Security
- Supportability
- Testability
- Usability

Quality is value to some person -- Jerry Weinberg

Risk

Hazard:

A dangerous condition (something that could trigger an accident)

Risk:

Possibility of suffering loss or harm.

Accident:

A hazard is encountered, resulting in loss or harm.

- 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

- 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)

Risk-Based Testing

- Two key dimensions:
  - Find errors (risk-based approach to technical tasks of testing)
  - Manage the process of finding errors (risk-based test management)
- Our focus today is on methods for finding errors efficiently.

Risks: Where to look for errors

- **Qualities**: Failure to conform to a quality criterion (risk of unreliability, risk of unmaintainability, etc.)
- **New things**: newer features may fail.
- **New technology:** new concepts lead to new mistakes.
- **New markets:** A different customer base will see and use the product differently.
- **Learning Curve:** mistakes due to ignorance.
- **Changed things**: changes may break old code.
- **Late changes:** 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.

Risks: Where to look for errors

- **Poor design or unmaintainable implementation.** Some internal design decisions make the code so hard to maintain that fixes consistently cause new problems.
- **Tired programmers:** long overtime over several weeks or months yields inefficiencies and 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.

Risks: Where to look for errors

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

Risks: Where to look for errors

- **Complexity**: complex code may be buggy.
- **Bugginess:** features with many known bugs may also have many unknown bugs.
- **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.

Risks: Where to look for errors

- **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.
- **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*

Risks: Where to look for errors

- **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.

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.*

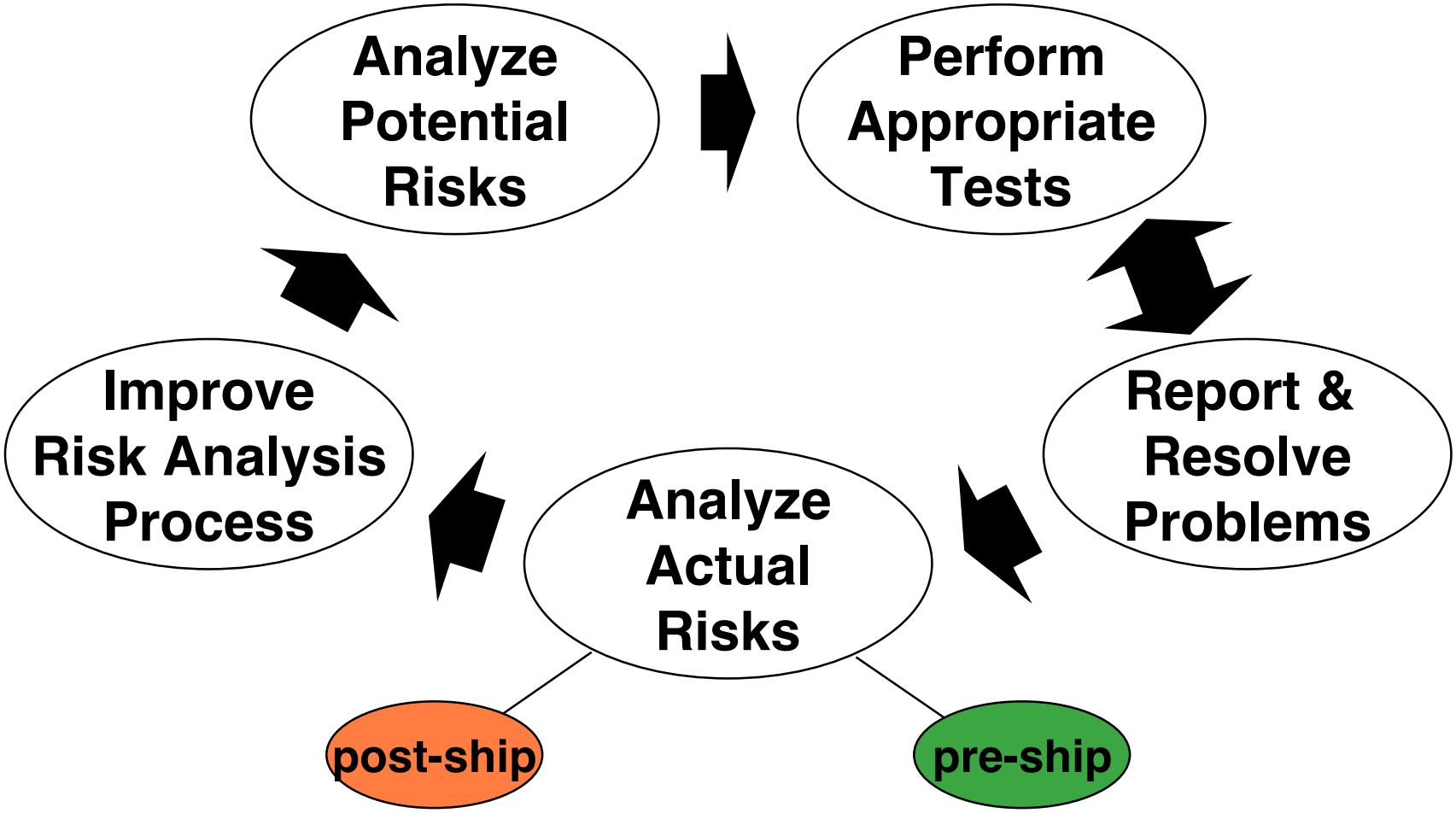
Build Your Own Model of Bug Patterns

- 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-Driven Testing Cycle



Test Case Design

- If the purpose of testing is to gain information about the product, then a test case's function is to elicit information quickly and efficiently.
- In information theory, we define "information" in terms of reduction of uncertainty. If there is little uncertainty, there is little information to be gained.
- A test case that promises no information is poorly designed. A good test case will provide information of value whether the program passes the test or fails it.

Thinking About Test Techniques

- Analyze the situation.
- Model the test space.
- Select what to cover.
- Determine test oracles.
- Configure the test system.
- Operate the test system.
- Observe the test system.
- Evaluate the test results.

A *test technique* is a recipe for performing these tasks that will reveal something worth reporting

Thinking About Test Techniques

- What is the difference between
  - User testing?
  - Usability testing?
  - User interface testing?

Thinking About Test Techniques

- Testing combines techniques that focus on:
  - ***Testers:*** ***who*** **does the testing.**
  - ***Coverage****:* ***what*** **gets tested.**
  - ***Potential problems****:* ***why*** **you're testing (what risk you're testing for).**
  - ***Activities****:* ***how*** **you test.**
  - ***Evaluation****:* ***how to tell whether the test passed or failed.***
- *All testing involves all five dimensions*.
- A technique focuses your attention on one or a few dimensions, leaving the others open to your judgment. You can combine a technique focused on one dimension with techniques focused on the other dimensions to achieve the result you want.

Thinking About Test Techniques

- **Examples**
  - **Testers:**
    - User testing; Beta testing; Subject-matter experts
  - **Coverage:**
    - Function testing; Domain testing; State-based testing; Path testing; Statement coverage; Configuration coverage
  - **Potential problems:**
    - Input / output / computation / storage constraints; Risk-based testing
  - **Activities:**
    - Exploratory testing; Scenario testing; Load testing; Performance testing
  - **Evaluation:**
    - Oracle-based testing; Comparison with saved results
- These examples are not definitive—how you classify a testing approach depends on what you think is most central to it. For example, is load testing problem oriented (denial of service) or activity oriented?
- The important thing is to conscious manage the 5 dimensions.

General Test Techniques

- Function
- Regression
- Domain driven
- Stress driven
- Specification driven
- Risk driven
- Scenario / use case / transaction flow
- User testing
- Exploratory
- Random / statistical

All of these have been used as ***the dominant technique*** in some companies.

How can approaches so different yield good overall results?

- We think that the answer is that each of these fixes only one of the dimensions for testing techniques.
- For example, function testing speaks to coverage but not to testers, risks, activities, or evaluation. You can vary all four of these and still be doing function testing.

General Test Techniques

- We provide an appendix that describes the 10 general test techniques that we listed on the previous slide.
- We aren't going to work through that appendix (or not in much detail) in this workshop, but these notes may be helpful for self-study, to fill in some of the details that we're skipping here.

Test Strategy

- "How we plan to cover the product so as to develop an adequate assessment of quality."
- A good test strategy is:
  - *Diversified*
  - *Specific*
  - *Practical*
  - *Defensible*

Test Strategy

- Makes use of test techniques.
- May be expressed by test procedures and cases.
- Not to be confused with test *logistics*, which involve the details of bringing resources to bear on the test strategy at the right time and place.
- You don’t have to know the entire strategy in advance. The strategy can change as you learn more about the product and its problems.

Test Cases/Procedures

- Test cases and procedures should manifest the test strategy.
- If your strategy is to “execute the test suite I got from Joe Third-Party”, how does that answer the prime strategic questions:
  - How will you *cover the product* and *assess quality*?
  - How is that *practical* and *justified* with respect to the *specifics* of this project and product?
- If you don’t know, then your real strategy is that you’re trusting things to work out.

Diverse Half-Measures

- There is no single technique that finds all bugs.
- We can't do any technique perfectly.
- We can't do all conceivable techniques.

Use **"diverse half-measures"--** lots of different points of view, approaches, techniques, even if no one strategy is performed completely.

Test Plan Components

- The following slides give examples of several charts, tables, etc.
- You probably won't have enough time to create all the documentation that would be useful. Treat these materials as ***optional***.
- Use the components that you find most useful to:
  - Clarify your own thinking
  - Communicate your thinking to others
  - Track your work or the work of someone else

Basic Test Documentation Components

Lists:

- Such as lists of fields, error messages, DLLs

Outlines: An outline organizes information into a hierarchy of lists and sublists

- Such as the testing objectives list later in the course notes

Tables: A table organizes information in two dimensions showing relationships between variables.

- Such as boundary tables, decision tables, combination test tables

Matrices: A matrix is a special type of table used for data collection.

- Such as the numeric input field matrix, configuration matrices
  - Refer to Testing Computer Software, pages 217-241. For more examples, see page Testing Computer Software, page 218.

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

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.

Myers' Boundary Table

Variable	Valid Case Equivalence Classes	Invalid Case Equivalence Classes	Boundaries and Special Cases	Notes
First number	-99 to 99	> 99 < -99 non-number expressions	99, 100 -99, -100 / : 0 null entry	
Second number	same as first	same as first	same	
Sum	-198 to 198			Are there other sources of data for this variable? Ways to feed it bad data?

Revised Boundary Analysis Table

Variable	Equivalence Class	Alternate Equivalence Class	Boundaries and Special Cases	Notes
First number	-99 to 99 digits	> 99 < -99 non-digits expressions	99, 100 -99, -100 /, 0, 9, : leading spaces or 0s null entry	
Second number	same as first	same as first	same	
Sum	-198 to 198 -127 to 127	??? -198 to –128 128 to 198	??? 127, 128, -127, -128	Are there other sources of data for this variable? Ways to feed it bad data?

Note that we've dropped the issue of "valid" and "invalid." This lets us generalize to partitioning strategies that don't have the **concept** of "valid" -- for example, **printer** equivalence classes.

Equivalence Classes: A Broad Concept

The notion of equivalence class is much broader than numeric ranges. Here are some examples:

- Membership in a common group
  - such as employees vs. non-employees. (Note that not all classes have shared boundaries.)
- Equivalent hardware
  - such as compatible modems
- Equivalent event times
  - such as before-timeout and after
- Equivalent output events
  - perhaps any report will do to answer a simple the question: Will the program print reports?
- Equivalent operating environments
  - such as French & English versions of Windows 3.1

Variables Well Suited to Equivalence Class Analysis

Many types of variables, including input, output, internal, hardware and system software configurations, and equipment states can be subject to equivalence class analysis. Here are some examples:

- ranges of numbers
- character codes
- how many times something is done
  - (e.g. shareware limit on number of uses of a product)
  - (e.g. how many times you can do it before you run out of memory)
- how many names in a mailing list, records in a database, variables in a spreadsheet, bookmarks, abbreviations
- size of the sum of variables, or of some other computed value (think binary and think digits)
- size of a number that you enter (number of digits) or size of a character string
- size of a concatenated string
- size of a path specification
- size of a file name
- size (in characters) of a document
- size of a file (note special values such as exactly 64K, exactly 512 bytes, etc.)
- size of the document on the page (compared to page margins) (across different page margins, page sizes)

Variables Well Suited to Equivalence Class Analysis

- size of a document on a page, in terms of the memory requirements for the page. This might just be in terms of resolution x page size, but it may be more complex if we have compression.
- equivalent output events (such as printing documents)
- amount of available memory (> 128 meg, > 640K, etc.)
- visual resolution, size of screen, number of colors
- operating system version
- variations within a group of “compatible” printers, sound cards, modems, etc.
- equivalent event times (when something happens)
- timing: how long between event A and event B (and in which order--races)
- length of time after a timeout (from JUST before to way after) -- what events are important?
- speed of data entry (time between keystrokes, menus, etc.)
- speed of input--handling of concurrent events
- number of devices connected / active
- system resources consumed / available (also, handles, stack space, etc.)
- date and time
- transitions between algorithms (optimizations) (different ways to compute a function)
- most recent event, first event
- input or output intensity (voltage)
- speed / extent of voltage transition (e.g. from very soft to very loud sound)

Using Test Matrices for Routine Issues

- After testing a simple numeric input field a few times, you may prefer a test matrix to present the same tests more concisely.
- Use a test matrix to show/track a series of test cases that are fundamentally similar.
  - For example, for most input fields, you'll do a series of the same tests, checking how the field handles boundaries, unexpected characters, function keys, etc.
  - As another example, for most files, you'll run essentially the same tests on file handling.
- The matrix is a concise way of showing the repeating tests.
  - Put the objects that you're testing on the rows.
  - Show the tests on the columns.
  - Check off the tests that you actually completed in the cells.

Reusable Test Matrix

Numeric Input Field													
		Nothing	LB of value	UB of value	LB of value - 1	UB of value + 1	0	Negative	LB number of digits or chars	UB number of digits or chars	Empty field (clear the default value)	Outside of UB number of digits or chars	Non-digits

Examples of integer-input tests

- Nothing
- Valid value
- At LB of value
- At UB of value
- At LB of value - 1
- At UB of value + 1
- Outside of LB of value
- Outside of UB of value
- 0
- Negative
- At LB number of digits or chars
- At UB number of digits or chars
- Empty field (clear the default value)
- Outside of UB number of digits or chars
- Non-digits
- Wrong data type (e.g. decimal into integer)
- Expressions
- Space
- Non-printing char (e.g., Ctrl+char)
- DOS filename reserved chars (e.g., "\ * . :")
- Upper ASCII (128-254)
- Upper case chars
- Lower case chars
- Modifiers (e.g., Ctrl, Alt, Shift-Ctrl, etc.)
- Function key (F2, F3, F4, etc.)

Error Handling when Writing a File

- full local disk
- almost full local disk
- write protected local disk
- damaged (I/O error) local disk
- unformatted local disk
- remove local disk from drive after opening file
- timeout waiting for local disk to come back online
- keyboard and mouse I/O during save to local disk
- other interrupt during save to local drive
- power out during save to local drive
- full network disk
- almost full network disk
- write protected network disk
- damaged (I/O error) network disk
- remove network disk after opening file
- timeout waiting for network disk
- keyboard / mouse I/O during save to network disk
- other interrupt during save to network drive
- local power out during save to network
- network power during save to network

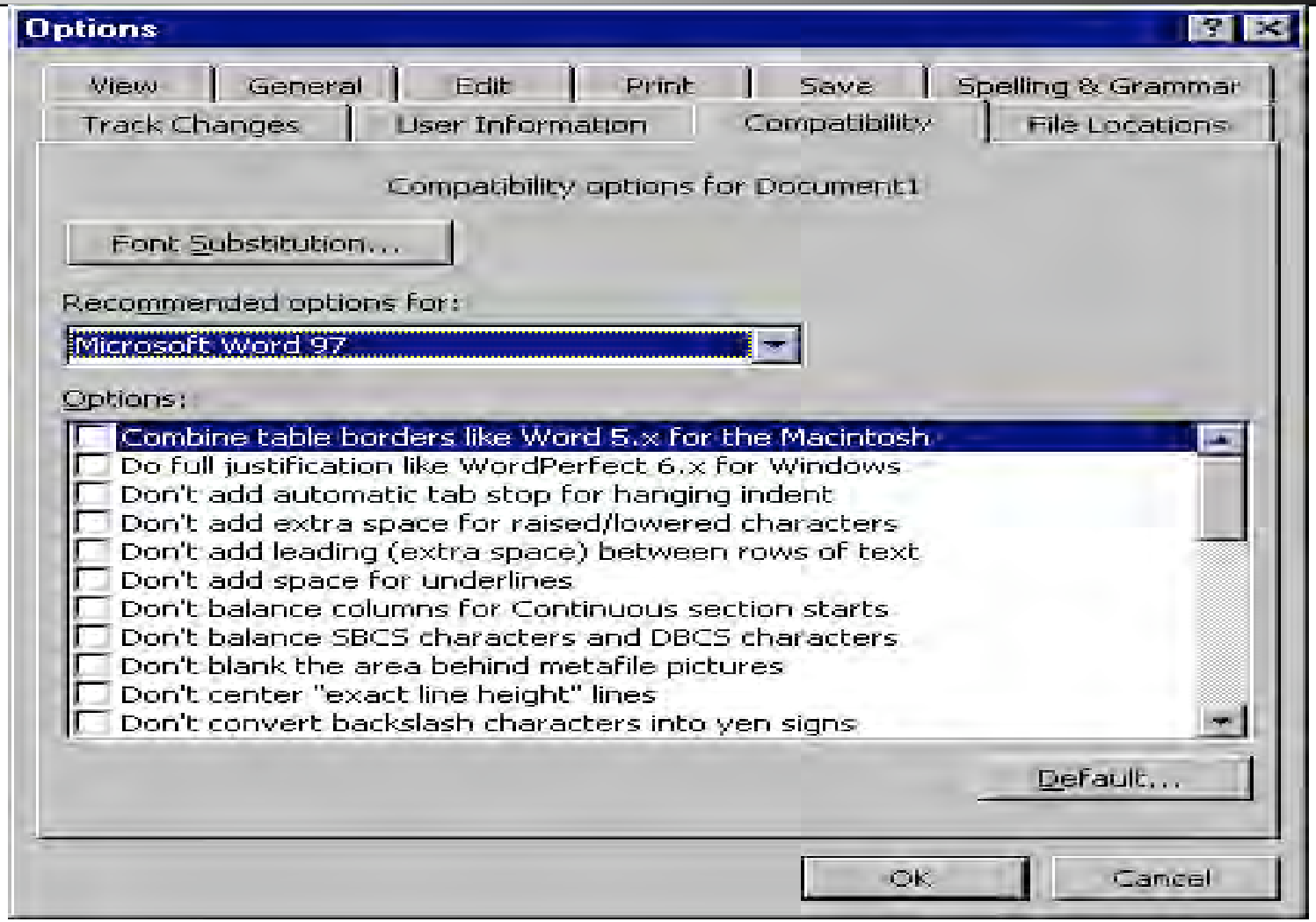
Routine Case Matrices

- **You can often re-use a matrix like this across products and projects.**
- **You can create matrices like this for a wide range of problems. Whenever you can specify multiple tests to be done on one class of object, and you expect to test several such objects, you can put the multiple tests on the matrix.**
- **Mark a cell green if you ran the test and the program passed it. Mark the cell red if the program failed.**
- **Write the bug number of the bug report for this bug.**
- **Write (in the cell) the automation number or identifier or file name if the test case has been automated.**

Routine Case Matrices

- Problems?
  - **What if your thinking gets out of date? (What if this program poses new issues, not covered by the standard tests?)**
  - **Do you need to execute every test every time? (or ever?)**
  - **What if the automation ID number changes? -- We still have a maintenance problem but it is not as obscure.**

Complex Data Relationships



Tabular Format for Data Relationships

Field	Entry source	Display	Print	Related variable	Relationship

Tabular Format for Data Relationships

Once you identify two variables that are related, test them together using boundary values of each or pairs of values that will trigger some other boundary.

- **This is not the most powerful process for looking at relationships. An approach like Cause-Effect Graphing is more powerful, if you have or can build a complete specification.**
- **I started using this chart as an *exploratory* tool for simplifying my look at relationships in overwhelmingly complex programs. (There doesn't have to be a lot of complexity to be "overwhelming.")**

Tabular Format for Data Relationships

- THE TABLE'S FIELDS

Field: *Create a row for each field (`Consultant`, `End Date`, and `Start Date` are examples of fields.)*

Entry Source: *What dialog boxes can you use to enter data into this field? Can you import data into this field? Can data be calculated into this field? List every way to fill the field -- every screen, etc.*

Display: *List every dialog box, error message window, etc., that can display the value of this field. When you re-enter a value into this field, will the new entry show up in each screen that displays the field? (Not always -- sometimes the program makes local copies of variables and fails to update them.)*

Print: *List all the reports that print the value of this field (and any other functions that print the value).*

Related to: *List every variable that is related to this variable. (What if you enter a legal value into this variable, then change the value of a constraining variable to something that is incompatible with this variable's value?)*

Relationship: *Identify the relationship to the related variable.*

Tabular Format for Data Relationships

Many relationships among data:

- **Independence**
  - **Varying one has no effect on the value or permissible values of the other.**
- **Causal determination**
  - **By changing the value of one, we determine the value of the other.**
  - **For example, in MS Word, the extent of shading of an area depends on the object selected. The shading differs depending on Table vs. Paragraph.**
- **Constrained to a range**
  - **For example, the width of a line has to be less than the width of the page.**
  - **In a date field, the permissible dates are determined by the month (and the year, if February).**
- **Selection of rules**
  - **Example, hyphenation rules depend on the language you choose.**

Tabular Format for Data Relationships

Many relationships among data:

- **Logical selection from a list**
  - **processes the value you entered and then figures out what value to use for the next variable. Example: timeouts in phone dialing:**
    - **0 on complete call 555-1212 but 95551212?**
    - **10 on ambiguous completion, 955-5121**
    - **30 seconds incomplete 555-121**
- **Logical selection *of* a list:**
  - **For example, in printer setup, choose:**
    - **OfficeJet, get Graphics Quality, Paper Type, and Color Options**
    - **LaserJet 4, get Economode, Resolution, and Half-toning.**

Look at Marick *(Craft of Software Testing)* for discussion of catalogs of tests for data relationships.

Data Relationship Table

- Looking at the Word options, you see the real value of the data relationships table. Many of these options have a lot of repercussions.
- You might analyze all of the details of all of the relationships later, but for now, it is challenging just to find out what all the relationships ARE.
- The table guides exploration and will surface a lot of bugs.
- ----------
- PROBLEM
- Works great for this release. Next release, what is your support for more exploration?

Configuration Planning Table

	Var 1	Var 2	Var 3	Var 4	Var 5
Config 1	V1-1	V2-1	V3-1	V4-1	V5-1
Config 2	V1-2	V2-2	V3-2	V4-2	V5-2
Config 3	V1-3	V2-3	V3-3	V4-3	V5-3
Config 4	V1-4	V2-4	V3-4	V4-4	V5-4
Config 5	V1-5	V2-5	V3-5	V4-5	V5-5
Config 6	V1-6	V2-6	V3-6	V4-6	V5-6

This table defines 6 standard configurations for testing. In later tests, the lab will set up a Config-1 system, a Config-2 system, etc., and will do its compatibility testing on these systems. The variables might be software or hardware choices. For example, Var 1 could be the operating system (V1-1 is Win 2000, V1-2 is Win ME, etc.) Var 2 could be how much RAM on the computer under test (V2-1 is 128 meg, V2-2 is 32 meg., etc.). Var 3 could be the type of printer, Var 4 the machine's language setting (French, German, etc.). Config planning tables are often filled in using the All Pairs algorithm.

Configuration Test Matrix

	Config 1	Config 2	Config 3	Config 4	Config 5	Config 6
Test 1	Pass	Pass	Pass	Pass	Pass	
Test 2		Fail	Pass	Pass	Pass	
Test 3	Pass	Pass	Pass	Pass	Pass	
Test 4	Pass	Fail	Fail		Pass	
Test 5	Fail	Pass	Fail	Pass	Pass	

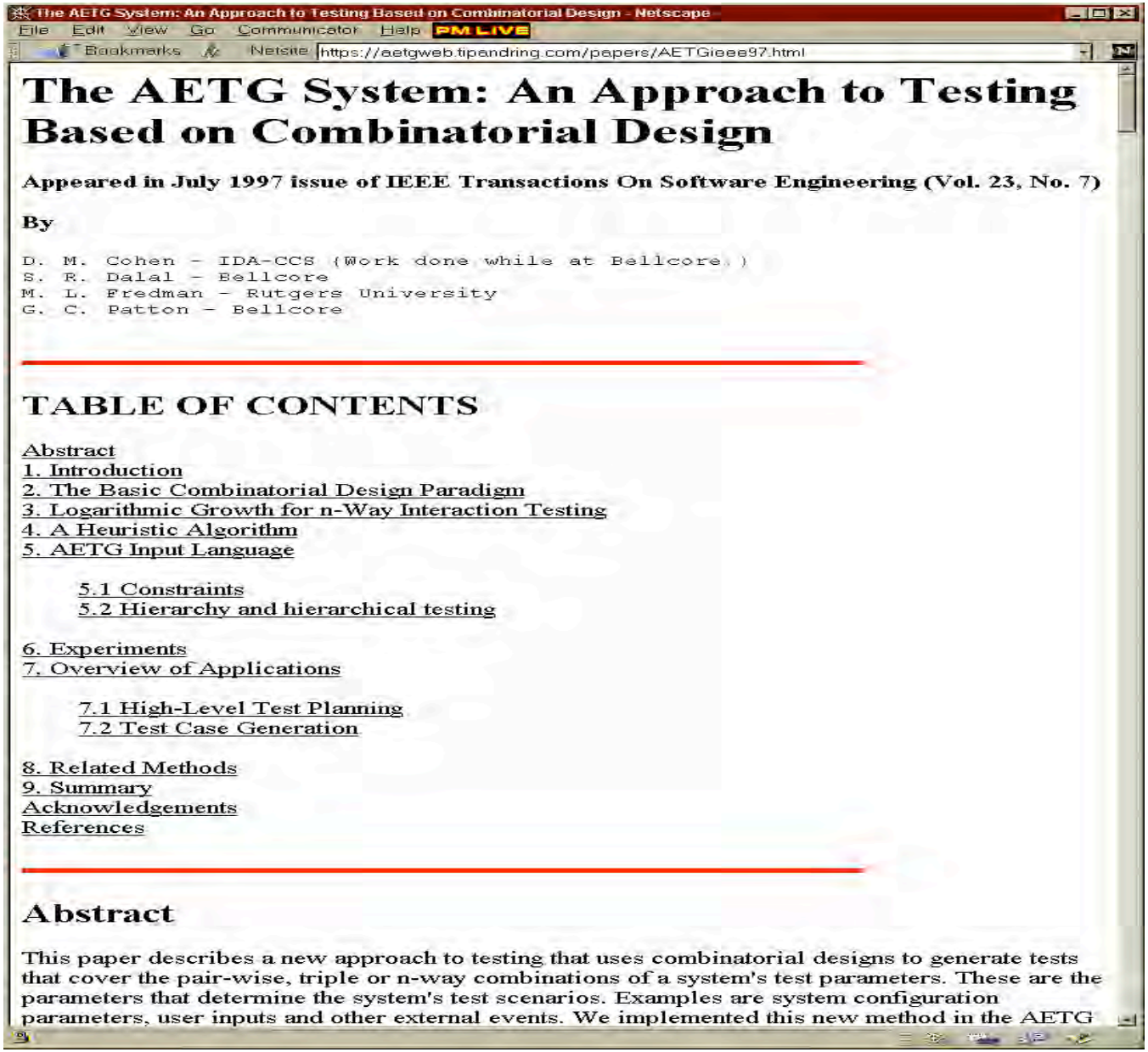
This matrix records the results of a series of tests against the 6 standard configurations that we defined in the Configuration Planning Table.

In this table, Config 1 has passed 3 tests, failed 1, and hasn't yet been tested with Test 2. Config 6 is still untested.

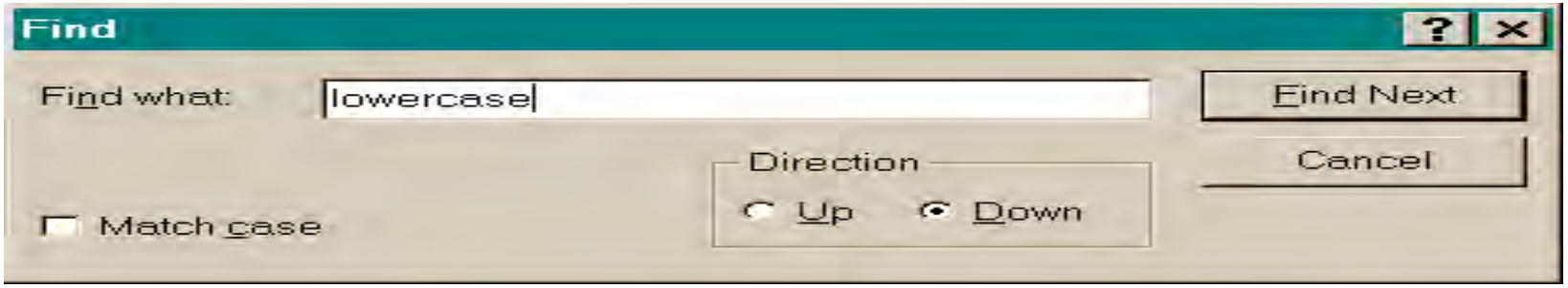
Testing Variables in Combination

Interesting papers.

- Cohen, Dalal, Parelius, Patton,"The Combinatorial Design Approach to Automatic Test Generation",IEEE Software, Sept. 96 http://computer.org:80/software/so1996/s5toc.htm
- Cohen, Dalal, Fredman, Patton, "The AETG System: An Approach to Testing Based on Combinatorial Design", IEEE Trans on SW Eng. Vol 23#7, July 97 http://computer.org:80/tse/ts1997/e7toc.htm
- OnLine requires IEEE membership for free access. See http://www.computer.org/epub/
- **Several other papers on AETG are available at https://aetgweb.tipandring.com/AboutAETGweb.html**
- Also interesting: http://www.stsc.hill.af.mil/CrossTalk/1997/oct/planning.html
- Jorgenson, Software Testing: A Craftsman's Approach
- Brian Marick, "Multi-Generating test ideas from expressions with booleans and relational operators" http://www.testing.com/tools/multi/README.html



Combinations Exercise / Illustration



- Here is a simple Find dialog. It takes three inputs:
  - `Find what:` a text string
  - `Match case:` yes or no
  - `Direction:` up or down
- Simplify this by considering only three values for the text string, “lowercase” and “Mixed Cases” and “CAPITALS”.
- (Note: To do a better job, we’d also choose input documents that would yield a “find” and a “don’t find” for each case. The input document would be another variable or, really, the intended result (Find / Don’t) would be the variable. We’ll think about that again after the exercise.)

Combinations Exercise

1 How many combinations of these three variables are possible?

2 List ALL the combinations of these three variables.

3 Now create combination tests that cover all possible pairs of values, but don't try to cover all possible triplets. List one such set.

4 How many test cases are in this set?

Combination Testing

- Imagine a program with 3 variables, V1 has 3 possible values, V2 has 2 possible values and V3 has 2 possible values.
- If V1 and V2 and V3 are independent, the number of possible combinations is 12 (3 x 2 x 2)
- Building a simple combination table:
  - Label the columns with the variable names, listing variables in descending order (of number of possible values)
  - Each column (before the last) will have repetition. Suppose that A, B, and C are in column K of N columns. To determine how many times (rows in which) to repeat A before creating a row for B, multiply the number of variable values in columns K+1, K+2, . . ., N.

Combination Testing

- Building an all-pairs combination table:
  - Label the columns with the variable names, listing variables in descending order (of number of possible values)
  - If the variable in column 1 has V1 possible values and the variable in column 2 has V2 possible values, then there will be at least V1 x V2 rows (draw the table this way but leave a blank row or two between repetition groups in column 1).
  - Fill in the table, one column at a time. The first column repeats each of its elements V2 times, skips a line, and then starts the repetition of the next element. For example, if variable 1's possible values are A, B, C and V2 is 2, then column 1 would contain A, A, blank row, B, B, blank row, C, C, blank row.

Combination Testing

- Building an all-pairs combination table:
  - In the second column, list all the values of the variable, skip the line, list the values, etc. For example, if variable 2's possible values are X,Y, then the table looks like this so far

A	X
A	Y
B	X
B	Y
C	X
C	Y

Combination Testing

- Building an all-pairs combination table:
  - Each section of the third column (think of AA as defining a section, BB as defining another, etc.) will have to contain every value of variable 3. Order the values such that the variables also make all pairs with variable 2.
  - Suppose variable 2 can be 1,0
  - The third section can be filled in either way, and you might highlight it so that you can reverse it later. The decision (say 1,0) is arbitrary.

A	X	1
A	Y	0
B	X	0
B	Y	1
C	X	
C	Y	

- *Now that we've solved the 3-column exercise, let's try adding more variables. Each of them will have two values.*

Combination Testing

- The 4th column went in easily (note that we started by making sure we hit all pairs of values of column 4 and column 2, then all pairs of column 4 and column 3.
- Watch this first attempt on column 5. We achieve all pairs of GH with columns 1, 2, and 3, but miss it for column 4.
- The most recent arbitrary choice was HG in the 2nd section. (Once that was determined, we picked HG for the third in order to pair H with a 1 in the third column.)
- So we will erase the last choice and try again:

A	X	1	E	G
A	Y	0	F	H
B	X	0	F	H
B	Y	1	E	G
C	X	1	F	H
C	Y	0	E	G

Combination Testing

•We flipped the last arbitrary choice (column 5, section 2, to GH from HG) and erased section 3. We then fill in section 3 by checking for missing pairs. GH, GH gives us two XG, XG pairs, so we flip to HP for the third section and have a column 2 X with a column 5 H and a column 2 Y with a column 5 G as needed to obtain all pairs.

A	X	1	E	G
A	Y	0	F	H
B	X	0	F	G
B	Y	1	E	H
C	X	1	F	H
C	Y	0	E	G

Combination Testing

•But when we add the next column, we see that we just can't achieve all pairs with 6 values. The first one works up to column 4 but then fails to get pair EJ or FI. The next fails on GJ, HI

A	X	1	E	G	I
A	Y	0	F	H	J
B	X	0	F	G	J
B	Y	1	E	H	I
C	X	1	F	H	J
C	Y	0	E	G	I

A	X	1	E	G	I
A	Y	0	F	H	J
B	X	0	F	G	I
B	Y	1	E	H	J
C	X	1	F	H	J
C	Y	0	E	G	I

Combination Testing

- When all else fails, add rows. We need one for GJ and one for HI, so add two rows. In general, we would need as many rows as the last column has values.
- The other values in the two rows are arbitrary, leave them blank and fill them in as needed when you add new columns. At the very end, fill the remaining blank ones with arbitrary values

A	X	1	E	G	I
A	Y	0	F	H	J
				G	J
B	X	0	F	G	I
B	Y	1	E	H	J
				H	I
C	X	1	F	H	J
C	Y	0	E	G	I

Combination Testing

- If a variable is continuous but maps to a number line, partition and use boundaries as the distinct values under test. If all variables are continuous, we end up with all pairs of all boundary tests of all variables. We don't achieve all triples, all quadruples, etc.
- If some combinations are of independent interest, add them to the list of n-tuples to test.
  - With the six columns of the example, we reduced 96 tests to 8. Give a few back (make it 12 or 15 tests) and you still get enormous reduction.
  - Examples of "independent interest" are known (from tech support) high risk cases, cases that jointly stress memory, configuration combinations (Var 1 is operating systems, Var 2 is printers, etc.) that are prevalent in the market, etc.

Charts: References

You can find plenty of example charts in Bill Perry's books, such as Effective Methods for Software Testing (2nd Ed., Wiley). Several of these will probably be useful, though (like the charts in these notes) you'll have to adapt them to your circumstances.

Heuristics from James Bach's Test Plan Evaluation Model

Heuristic	Basis for the Heuristic
1. Testing should be optimized to find important problems fast, rather than attempting to find all problems with equal urgency.	The later in the project that a problem is found, the greater the risk that it will not be safely fixed in time to ship. The sooner a problem is found after it is created, the lesser the risk of a bad fix.
2. Test strategy should focus most effort on areas of potential technical risk, while still putting some effort into low risk areas just in case the risk analysis is wrong.	Complete testing is impossible, and we can never know if our perception of technical risk is completely accurate.
3. Test strategy should address test platform configuration, how the product will be operated, how the product will be observed, and how observations will be used to evaluate the product.	Sloppiness or neglect within any of these four basic testing activities will increase the likelihood that important problems will go undetected.

Heuristics for Test Plan Evaluation

Heuristic	Basis for the Heuristic
4. Test strategy should be diversified in terms of test techniques and perspectives. Methods of evaluating test coverage should take into account multiple dimensions of coverage, including structural, functional, data, platform, operations, and requirements.	No single test technique can reveal all important problems in a linear fashion. We can never know for sure if we have found all the problems that matter. Diversification minimizes the risk that the test strategy will be blind to certain kinds of problems. ***Use diverse half-measures to go after low-hanging fruit.***
5. The test strategy should specify how test data will be designed and generated.	It is common for the test strategy to be organized around functionality or code, leaving it to the testers to concoct test data on the fly. Often that indicates that the strategy is too focused on validating capability and not focused enough on reliability.

Heuristics for Test Plan Evaluation

Heuristic	Basis for the Heuristic
6. Not all testing should be pre-specified in detail. The test strategy should incorporate reasonable variation and make use of the testers' ability to use situational reasoning to focuse on important, but unanticipated problems.	A rigid test strategy may make it more likely that a particular subset of problems will be uncovered, but in a complex system it reduces the likelihood that *all* important problems will be uncovered. Reasonable variability in testing, such as that which results from interactive, exploratory testing, increases incidental test coverage, without substantially sacrificing essential coverage.
7. It is important to test against implied requirements—the full extent of what the requirements mean, not just what they say.	Testing only against explicit written requirements will not reveal all important problems, since defined requirements are generally incomplete and natural language is inherently ambiguous.

Heuristics for Test Plan Evaluation

Heuristic	Basis for the Heuristic
8. The test project should promote collaboration with all other functions of the project, especially developers, technical support, and technical writing. Whenever possible, testers should also collaborate with actual customers and users, in order to better understand their requirements.	Other teams and stakeholders often have information about product problems or potential problems that can be of use to the test team. Their perspective may help the testers make a better analysis of risk. Testers may also have information that is of use to them.
9. The test project should consult with development to help them build a more testable product.	The likelihood that a test strategy will serve its purpose is profoundly affected by the testability of the product.

Heuristics for Test Plan Evaluation

Heuristic	Basis for the Heuristic
10. A test plan should highlight the non-routine, project-specific aspects of the test strategy and test project.	Virtually every software project worth doing involves special technical challenges that a good test effort must take into account. A completely generic test plan usually indicates a weak test planning process. It could also indicate that the test plan is nothing but unchanged boilerplate.

Heuristics for Test Plan Evaluation

Heuristic	Basis for the Heuristic
11. The test project should use humans for what humans do well and use automation for what automation does well. Manual testing should allow for improvisation and on the spot critical thinking, while automated testing should be used for tests that require high repeatability, high speed, and no judgment.	Many test projects suffer under the false belief that human testers are effective when they use exactingly specified test scripts, or that test automation duplicates the value of human cognition in the test execution process. Manual and automated testing are not two forms of the same thing. They are two entirely different classes of test technique.

Heuristics for Test Plan Evaluation

Heuristic	Basis for the Heuristic
12. The test schedule should be represented and justified in such a way as to highlight any dependencies on the progress of development, the testability of the product, time required to report problems, and the project team's assessment of risk.	A monolithic test schedule in a test plan often indicates the false belief that testing is an independent activity. The test schedule can stand alone only to the extent that the product the highly testable, development is complete, and the test process is not interrupted by the frequent need to report problems.
13. The test process should be kept off of the critical path to the extent possible. This can be done by testing in parallel with development work, and finding problems worth fixing faster than the developers fix them.	This is important in order to deflect pressure to truncate the testing process.

Heuristics for Test Plan Evaluation

Heuristic	Basis for the Heuristic
14. The feedback loop between testers and developers should be as tight as possible. Test cycles should be designed to provide rapid feedback to developers about recent additions and changes they have made before a full regression test is commenced. Whenever possible testers and developers should work physically near each other.	This is important in order to maximize the efficiency and speed of quality improvement. It also helps keep testing off of the critical path.

Heuristics for Test Plan Evaluation

Heuristic	Basis for the Heuristic
15. The test project should employ channels of information about quality other than formal testing in order to help evaluate and adjust the test project. Examples of these channels are inspections, field testing, or informal testing by people outside of the test team.	By examining product quality information gathered through various means beyond the test team, blind spots in the formal test strategy can be uncovered.
16. All documentation related to the test strategy, including test cases and procedures, should be undergo review by someone other than the person who wrote them. The review process used should be commensurate with the criticality of the document.	Tunnel-vision is the great occupational hazard of testing. Review not only helps to reveal blind spots in test design, but it can also help promote dialog and peer education about test practices.

Evaluating Your Plan: Context Free Questions

Based on: *The CIA's Phoenix Checklists (Thinkertoys, p. 140) and Bach's Evaluation Strategies (Rapid Testing Course notes)*

- Can you solve the whole problem? Part of the problem?
- What would you like the resolution to be? Can you picture it?
- How much of the unknown can you determine?
- What reference data are you using (if any)?
- What product output will you evaluate?
- How will you do the evaluation?
- Can you derive something useful from the information you have?
- Have you used all the information?
- Have you taken into account all essential notions in the problem?
- Can you separate the steps in the problem-solving process? Can you determine the correctness of each step?
- What creative thinking techniques can you use to generate ideas? How many different techniques?
- Can you see the result? How many different kinds of results can you see?
- How many different ways have you tried to solve the problem?

Evaluating Your Plan: Context Free Questions

- What have others done?
- Can you intuit the solution? Can you check the results?
- What should be done?
- How should it be done?
- Where should it be done?
- When should it be done?
- Who should do it?
- What do you need to do at this time?
- Who will be responsible for what?
- Can you use this problem to solve some other problem?
- What is the unique set of qualities that makes this problem what it is and none other?
- What milestones can best mark your progress?
- How will you know when you are successful?
- How conclusive and specific is your answer?

Appendix on General Test Techniques

- The following slides review 10 general test techniques. In previous talks, we've called these "paradigms" because many companies have organized their entire testing effort and testing thinking around one or two of them.
- We won't discuss many of these slides in the workshop, but we hope that these will be helpful reference materials to add some detail to comments that we make about these techniques in class.
- There is nothing magical about these techniques. They overlap. They don't collectively cover everything that would be good to do.
- Imagine that you are one of the people who has adopted one of these techniques as your primary approach, your paradigm:
  - What makes for an excellent test?
  - What is your approach best for?
  - What are some weaknesses in your approach?

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). Push it in as many ways as possible, as hard as possible.

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

GUI Regression: Interesting Papers

- 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

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

- In classical domain testing
  - Two values (single points or n-tuples) are equivalent if the program would take the same path in response to each.
- The classical domain strategies all assume
  - that the predicate interpretations are simple, linear inequalities.
  - the input space is continuous and
  - coincidental correctness is disallowed.
- It is possible to move away from these assumptions, but the cost can be high, and the emphasis on paths is troublesome because of the high number of possible paths through the program.
  - Clarke, Hassell, & Richardson, p. 388

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.*

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: 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

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: Interesting Papers

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

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.

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 may not 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.

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 will make a fuss 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.

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

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
  - OR prioritize the testing effort in terms of the relative risk of different areas or issues we could test for.
- Paradigmatic case(s)
  - Failure Mode and Effects Analysis (FMEA)
  - Equivalence class analysis, reformulated.
  - Test in order of frequency of use (Musa).
  - Stress tests, error handling tests, security tests, tests looking for predicted or feared errors, sample from predicted-bugs list.

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.
- We think this is only half of the risk story. The other half is focuses on test design.
  - It seems to us that 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?

Evaluating Risk

- 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 it to easy tests doesn't tell us much about what will happen to the program in the field.

Risk-Based Testing: Interesting Papers

- 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

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, the technical results are mixed. Distinguish marketing betas from technical betas.

Exploratory Testing

Simultaneously:

- Learn about the product
- Learn about the market
- Learn about the ways the product could fail
- Learn about the weaknesses of the product
- Learn about how to test the product
- Test the product
- Report the problems
- Advocate for repairs
- **Develop new tests based on what you have learned so far.**

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.

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.

Exploratory Testing: Interesting Papers

- Chris Agruss & Bob Johnson, Ad Hoc Software Testing Exploring the Controversy of Unstructured Testing
- Whittaker & Jorgenson, How to Break Software

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.
    - 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)
  - Some of us are still wrapping our heads around the richness of work in this field. 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

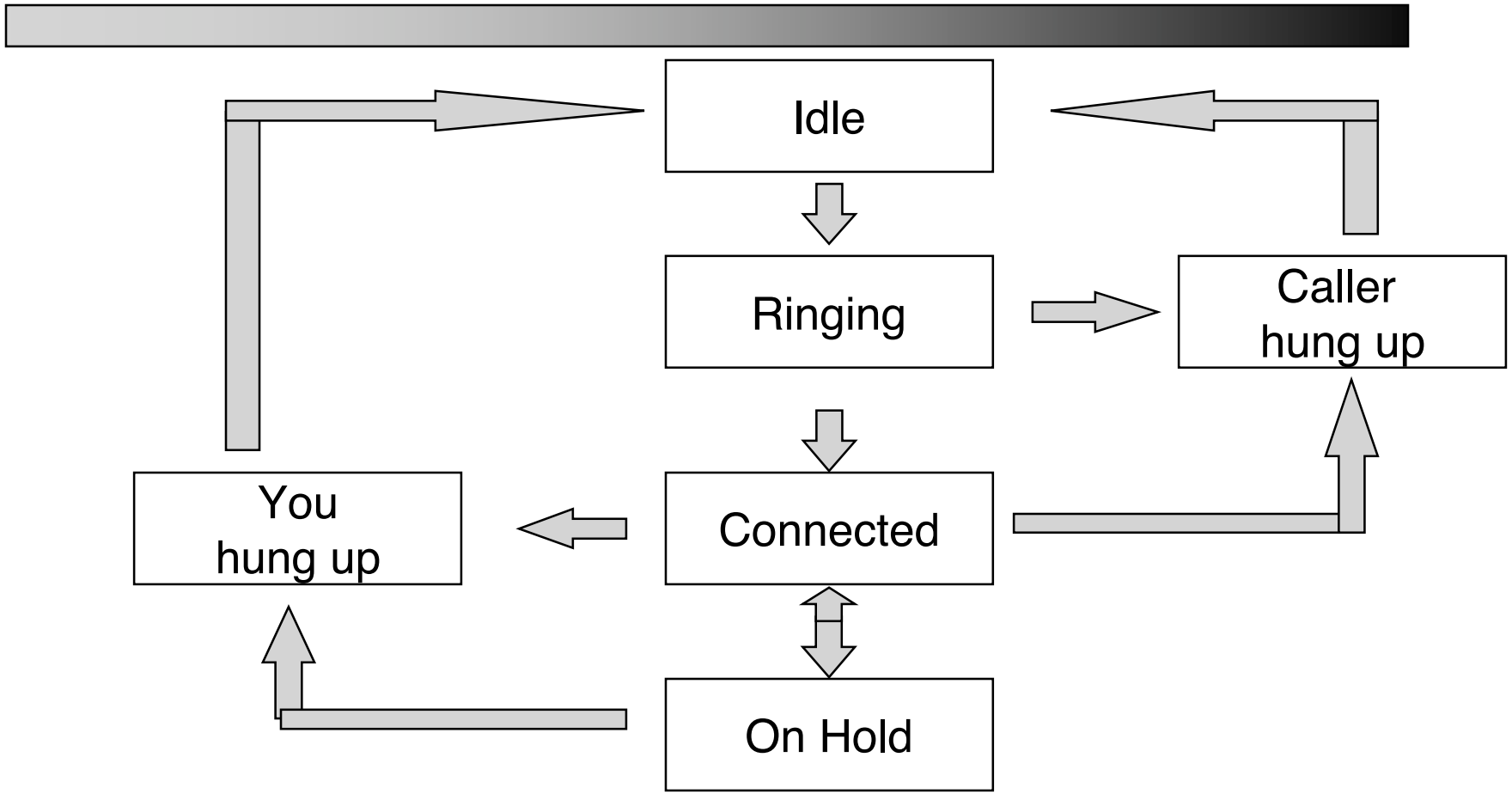
Random / Statistical Testing: Non-Stochastic

- 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 the achieve different results from the same set of inputs.
  - Other test using fully deterministic oracles (see discussion of oracles, below)
  - Other tests using heuristic oracles (see discussion of oracles, below)

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



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?"

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

Stochastic, assert or diagnostics-based random tests

- 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 Ti, 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—Ti 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 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)
  - I haven't done this kind of work. Here's what I understand:
    - 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 or walking a control panel on a printer)
    - In general, well suited to a limited-functionality client that will not be powered down or rebooted very often.
    - Maintenance is a critical issue because design changes add or subtract nodes, forcing a regeneration of the model.

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, _, _, q0, F) where M is the machine, Q is a finite set of states, _ is a finite set of inputs commonly called the "alphabet," _ is the transition function that maps Q x _ to Q,, q0 is one particular element of Q identified as the initial or stating state, and F ⊆ 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

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

> 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.
>
> 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.

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 have 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.
  - These methods will often cover many types of risks, but will obscure the need for other tests less 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: Interesting Papers

- 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.