

# The Nature of Exploratory Testing

by

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

Professor of Software Engineering

Florida Institute of Technology

and

James Bach

Principal, Satisfice Inc.

These notes are partially based on research that was supported by NSF Grant EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers." Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Kaner & Bach grant permission to make digital or hard copies of this work for personal or classroom use, including use in commercial courses, provided that (a) Copies are not made or distributed outside of classroom use for profit or commercial advantage, (b) Copies bear this notice and full citation on the front page, and if you distribute the work in portions, the notice and citation must appear on the first page of each portion. Abstracting with credit is permitted. The proper citation for this work is "Exploratory & Risk-Based Testing (2004) [www.testingeducation.org](http://www.testingeducation.org)", (c) Each page that you use from this work must bear the notice "Copyright (c) Cem Kaner and James Bach", or if you modify the page, "Modified slide, originally from Cem Kaner and James Bach", and (d) If a substantial portion of a course that you teach is derived from these notes, advertisements of that course should include the statement, "Partially based on materials provided by Cem Kaner and James Bach." To copy otherwise, to republish or post on servers, or to distribute to lists requires prior specific permission and a fee. Request permission to republish from Cem Kaner, [kaner@kaner.com](mailto:kaner@kaner.com).

# About Cem Kaner

My current job titles are

- Professor of Software Engineering
- Director of the Center for Software Testing Education at the Florida Institute of Technology, and
- Research Fellow at Satisfice, Inc.

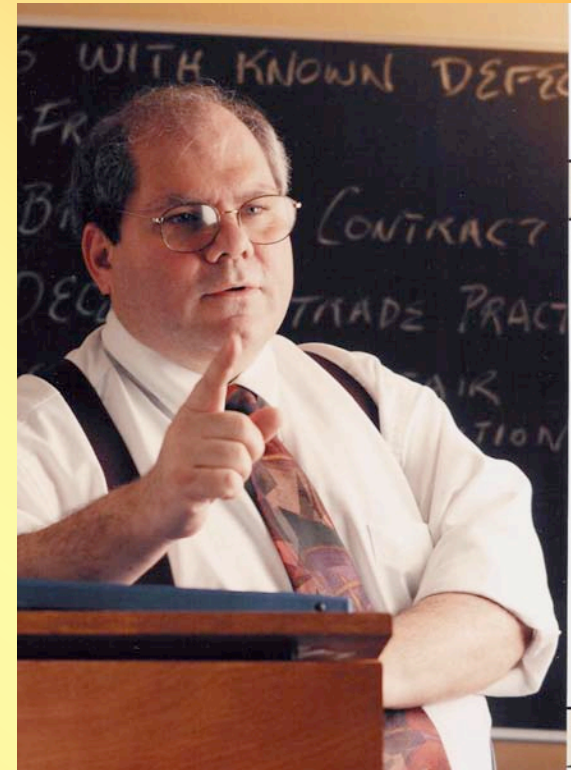
I'm also an attorney, whose work focuses on same theme as the rest of my career: *satisfaction and safety of software customers and workers*.

I've worked as a programmer, tester, writer, teacher, user interface designer, software salesperson, organization development consultant, as a manager of user documentation, software testing, and software development, and as an attorney focusing on the law of software quality. These have provided many insights into relationships between computers, software, developers, and customers.

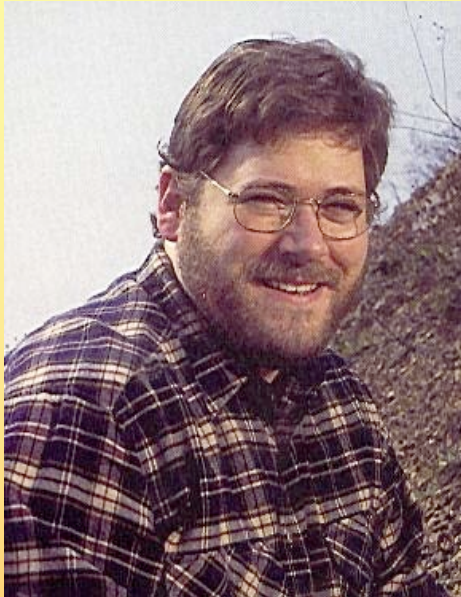
I'm the senior author of three books:

- *Lessons Learned in Software Testing* (with James & Bret Pettichord)
- *Bad Software* (with David Pels)
- *Testing Computer Software* (with Jack Falk & Hung Quoc Nguyen).

I studied Experimental Psychology for my Ph.D., with a dissertation on Psychophysics (essentially perceptual measurement). This field nurtured my interest in *human factors* (and thus the usability of computer systems) and in *measurement theory* (and thus, the development of valid software metrics.)



# About James Bach



I started in this business as a programmer. I like programming. But I find the problems of software quality analysis and improvement more interesting than those of software production. For me, there's something very compelling about the question "How do I know my work is good?" Indeed, how do I know anything is good? What does good mean? That's why I got into SQA, in 1987.

Today, I work with project teams and individual engineers to help them plan SQA, change control, and testing processes that allow them to understand and control the risks of product failure. I also assist in product risk analysis, test design, and in the design and implementation of computer-supported testing. Most of my experience is with market-driven Silicon Valley software companies like Apple Computer and Borland, so the techniques I've gathered and developed are designed for use under conditions of compressed schedules, high rates of change, component-based technology, and poor specification.

# What is Testing?

*A technical investigation  
done to expose  
quality-related information  
about the product  
under test*

*Let's try an exercise ...*



# Information Objectives: Which Group is Better?

## *Testing Group 1*

	Found pre-release
Function A	100
Function B	0
Function C	0
Function D	0
Function E	0
Total	100

## *Testing Group 2*

Function A	50
Function B	6
Function C	6
Function D	6
Function E	6
Total	74

From Marick,  
*Classic Testing  
Mistakes*

Suppose that:

Two groups test the same program.

- The functions are equally important
- The bugs are equally significant

*This is artificial, but it sets up a simple context for a discussion of tradeoffs.*

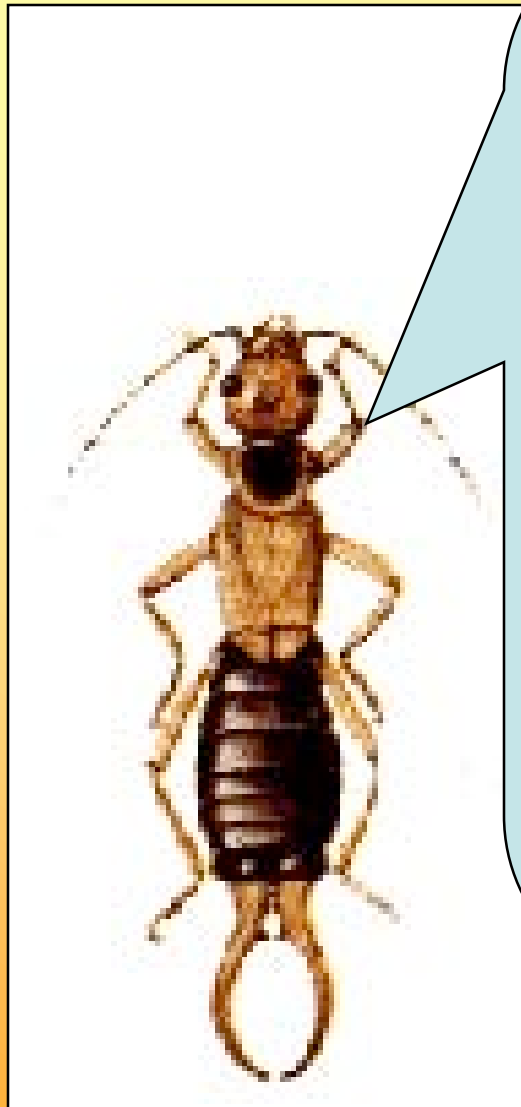
# Which group is better?

	Found pre-release	Found later	Total
Function A	100	0	100
Function B	0	12	12
Function C	0	12	12
Function D	0	12	12
Function E	0	12	12
Total	100	48	148
Function A	50	50	100
Function B	6	6	12
Function C	6	6	12
Function D	6	6	12
Function E	6	6	12
Total	74	74	148

# Information objectives

Different testing strategies serve different information objectives.

- Find defects
- Maximize bug count
- Block premature product releases
- Help managers make ship / no-ship decisions
- Minimize technical support costs
- Assess conformance to specification
- Conform to regulations
- Minimize safety-related lawsuit risk
- Find safe scenarios for use of the product
- Assess quality
- Verify correctness of the product
- Assure quality



**Exploratory  
testing  
is useful for  
discovering new  
information**



# 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

- Exploratory testing is not a testing technique. It's a way of thinking about testing.



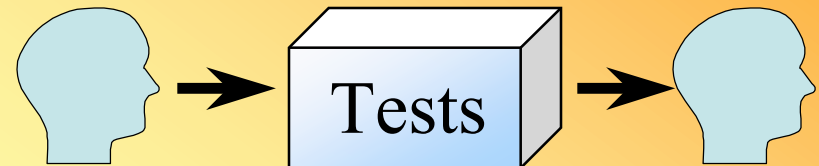
Any testing technique can be  
used in an exploratory way.

# Exploratory Testing

- Every competent tester does *some* exploratory testing.  
For example -- bug regression:  
Report a bug, the programmer claims she fixed the bug, so you test the fix.
  - Start by reproducing the steps you used in the bug report to expose the failure.
  - Then vary your testing to search for side effects.
    - These variations are not predesigned. This is an example of chartered exploratory testing.

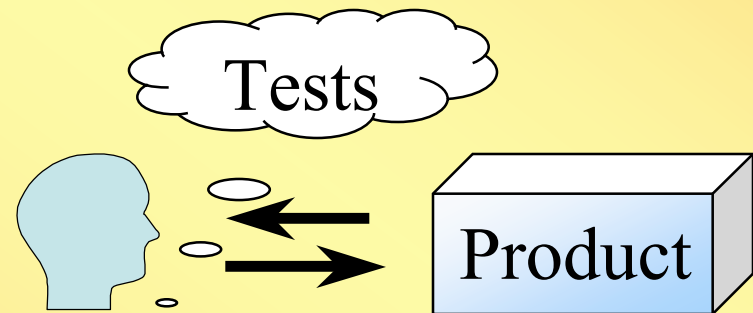
# Contrasting Approaches

In *scripted* testing, tests are first designed and recorded. Then they may be executed at some later time or by a different tester.



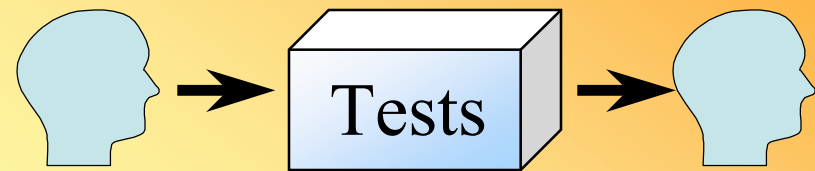
**vs.**

In *exploratory* testing, tests are designed and executed at the same time, and they often are not recorded.



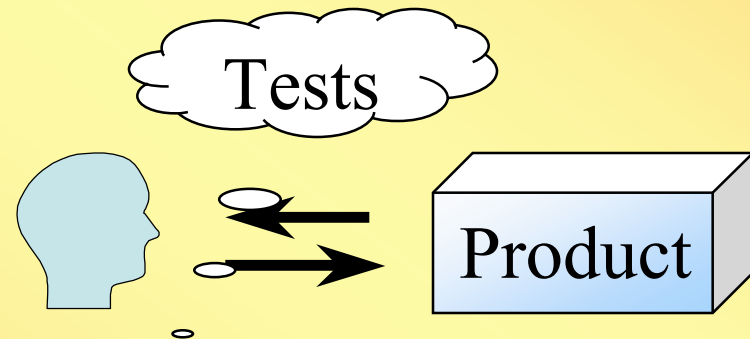
# Contrasting Approaches

Scripted testing emphasizes  
*accountability* and *decidability*.



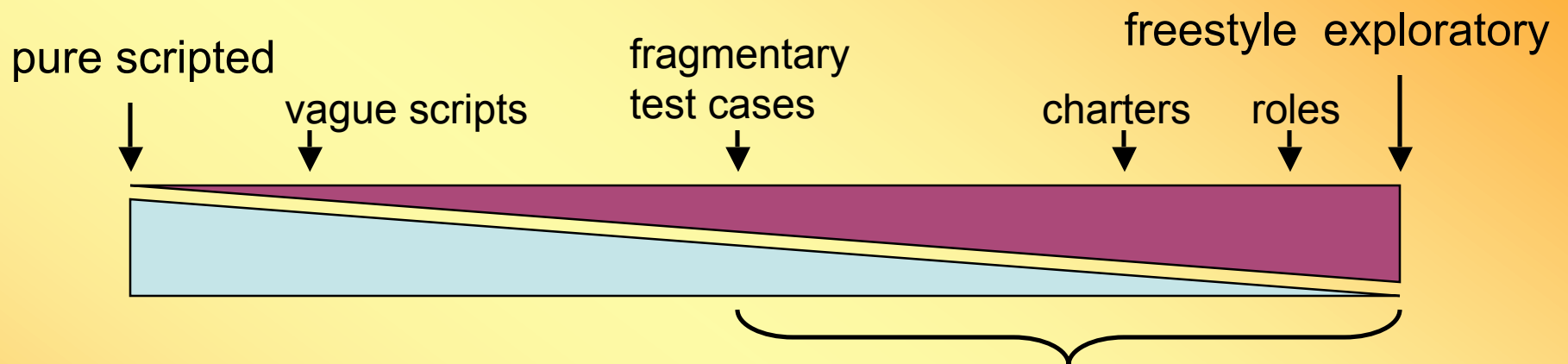
**vs.**

Exploratory testing emphasizes  
*adaptability* and *learning*.



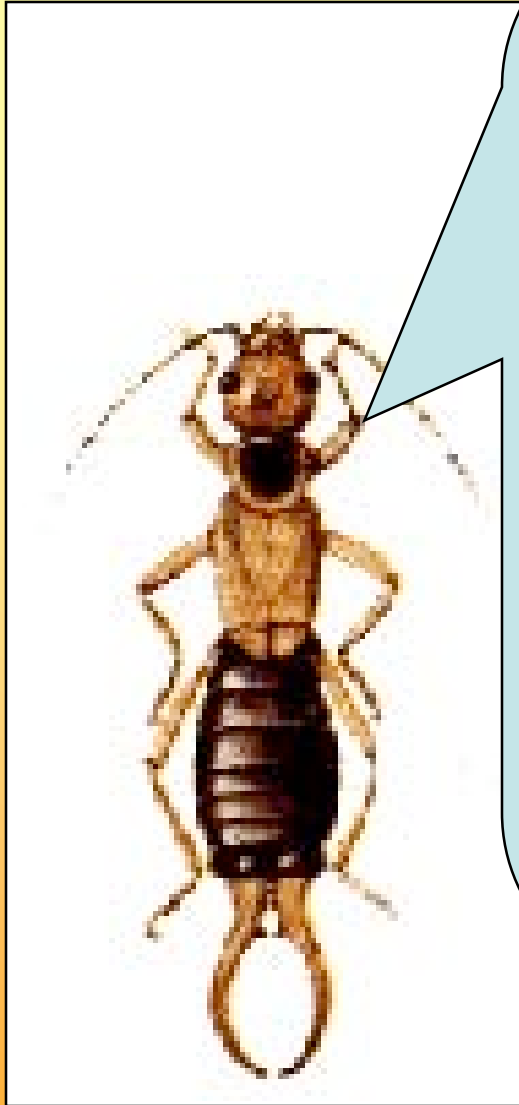


# Exploratory Testing Continuum



*When I say “exploratory testing” and don’t qualify it, I mean anything on the exploratory side of this continuum.*

Plenty of work is done  
between the extremes of the continuum.



**So how do we  
do the  
exploration?**

# Key challenges of exploratory testing

- Software testing poses several core challenges to the skilled practitioner / manager. The same challenges apply to exploratory testers:
  - **Learning** (How do we get to know the program?)
  - **Visibility** (How to see below the surface?)
  - **Control** (How to set internal data values?)
  - **Risk / selection** (Which are the best tests to run?)
  - **Execution** (What's the most efficient way to run the tests?)
  - **Logistics** (What environment is needed to support test execution?)
  - **The oracle problem** (How do we tell if a test result is correct?)
  - **Reporting** (How can we replicate a failure and report it effectively?)
  - **Documentation** (What test documentation do we need?)
  - **Measurement** (What metrics are appropriate?)
  - **Stopping** (How to decide when to stop testing?)

# Exploration & Learning

- This talk will focus on the learning strategies and tactics that testers use in the course of exploration.
- We can talk about the other challenges, and I can quickly put together a brief presentation on any of the others, but I think that the learning issues are the most challenging and the most interesting.

# Exploration and Learning

- Explorers use a variety of tactics to learn about the product and its context (market, environment, etc.)
- There are enormous individual and contextual differences
- Here are a few examples.
  - Early coverage heuristics
  - Active reading of reference materials
  - Attacks
  - Failure mode lists
  - Project risk lists
  - Scenario development
  - Models
- I've provided slides for each of these but won't attempt to cover all of them in class. The ones we skip are for your later reference.



# Early coverage heuristics

- We can use tests as a structure for learning about the basics of the program:
  - Test capability before reliability (*Start with function testing*)
  - Test single features and single variables before combinations (*Domain testing*)
  - Test visible / accessible features before hidden or esoteric ones (*Develop an appreciation of the designers' view of the mainstream*)
  - Test common use cases before unusual ones (*Develop an appreciation of how the product serves the target users' basic needs.*)

# Exploration & Learning

- Explorers use a variety of tactics to learn about the product and its context (market, environment, etc.)
- There are enormous individual and contextual differences
- Here are a few examples.
  - Early coverage heuristics

## – **Active reading of reference materials**

- Attacks
- Failure mode lists
- Project risk lists
- Scenario development
- Models

# Active Reading

- Exploratory testers, on approaching a product, often look for the following types of information:
  - **Affordances:** Ways the product can be used.
  - **Dimensions & Variables:** Product space and what changes within it.
  - **Relationships & Interactions:** functions that cooperate or interfere.
  - **Navigation:** Where things are and how to get to them.
  - **Benefits:** What the product is good for--when it has no bugs.
  - **Consistencies:** Fulfillment of logical, factual, and cultural expectations.
  - **Oracles:** Specific mechanisms or principles by which you can spot bugs.
  - **Bugs and Risks:** Specific problems and potential problems that matter.
  - **History:** Where the product has been, and how it came to be.
  - **Operations:** Its users and the conditions under which it will be used.
  - **Complexities & Challenges:** Discover the hardest things to test.
  - **Attitudes:** What your (testers') clients care about; what they want from you.
  - **Capabilities:** What the testing staff are good at; how they have excelled in the past.
  - **Resources:** Discover tools and information that might help you test.

# Ambiguity analysis

- There are many sources of ambiguity in software design and development.
  - In wording or interpretation of specifications or standards
  - In expected response of the program to invalid or unusual input
  - In behavior of undocumented features
  - In conduct and standards of regulators / auditors
  - In customers' interpretation of their needs and the needs of the users they represent
  - In definitions of compatibility among 3rd party products
- Whenever there is ambiguity, there is a strong opportunity for a defect (at least in the eyes of anyone who understands the world differently from the implementation).
  - Richard Bender teaches this well. If you can't take his course, you can find notes based on his work in Rodney Wilson's *Software RX: Secrets of Engineering Quality Software*
  - An interesting workbook: Cecile Spector, *Saying One Thing, Meaning Another*

# Active Reading

- Plenty of documents tell us about the product, its market, and its context.
- The active reader gathers information rather than passively receiving it.
- Active readers commonly
  - approach what they are reading with specific questions
  - summarize and reorganize what they read
  - describe what they've read to others
  - use a variety of other tactics to integrate what they are reading with what they already know or want to learn
- There are plenty of discussions / tutorials on how to do active reading on the net (just search google for “active reading.”)
- We provide several tips for active reading of specifications in our lecture notes, <http://www.testingeducation.org/k04/index.htm>



# Learning from available specifications

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

# Learning from available specifications

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

# Active reading

- These documents (explicit & implicit specs) tell us about
  - stakeholder / designer / programmer intent
  - customer expectations
  - user expectations
  - interoperability requirements and other third-party technical expectations
- Every statement of fact is a prediction about the product, that can be turned into a test
- Ambiguities, confusion, and errors in the company-written documents, and in other source documents the company has relied on, point to opportunities for software error (and thus for tests)
  
- As with the information we collect while running early tests, we use these documents to generate test ideas and testing notes (artefacts)

# Active reading

- As with the information we collect while running early tests, we use these documents to generate test ideas and testing notes (artefacts)
- Our artefacts might include
  - function lists
  - lists of variables and their use
  - lists of use cases
  - traceability matrices
- So how is this different from traditional test planning?
  - We create and execute tests as we read. The tests help us understand the documents (understand-by-example) along with exposing bugs in the product.
  - The intent of the research is development of personal insight, not development of artefacts. The artefacts support the individual explorer, but may or may not be of any use to the next reader (if they are shared at all)

# Exploration & Learning

- Explorers use a variety of tactics to learn about the product and its context (market, environment, etc.)
- There are enormous individual and contextual differences
- Here are a few examples.
  - Early coverage heuristics
  - Active reading of reference materials

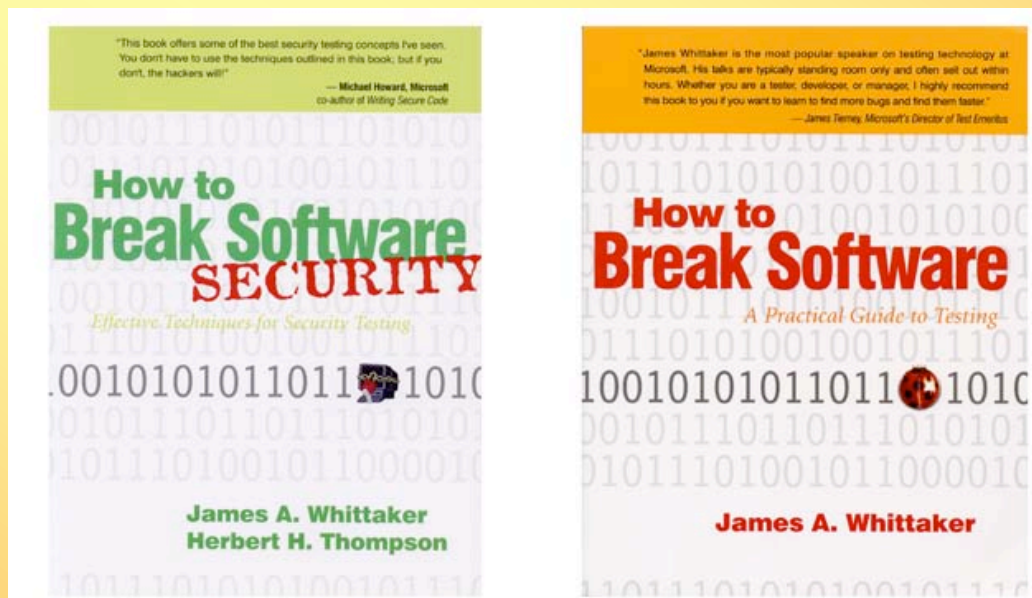
## – **Attacks**

- Failure mode lists
- Project risk lists
- Scenario development
- Models



# Attacks

Some errors are so common that there are well-known attacks for them.



An *attack* is a stereotyped class of tests, optimized around a specific type of error.

# Attacks

- James Whittaker and Alan Jorgensen pulled together a powerful collection of attacks (and examples).
- In his book, *How to Break Software*, Professor Whittaker expanded the list and, for each attack, discussed
  - When to apply it
  - What software errors make the attack successful
  - How to determine if the attack exposed a failure
  - How to conduct the attack, and
  - An example of the attack.
- We'll list *How to Break Software's* attacks here, but recommend the book's full discussion.

# *How to Break Software attacks*

- **User interface attacks: Exploring the input domain**
  - Attack 1: Apply inputs that force all the error messages to occur
  - Attack 2: Apply inputs that force the software to establish default values
  - Attack 3: Explore allowable character sets and data types
  - Attack 4: Overflow input buffers
  - Attack 5: Find inputs that may interact and test combinations of their values
  - Attack 6: Repeat the same input or series of inputs numerous times
- **User interface attacks: Exploring outputs**
  - Attack 7: Force different outputs to be generated for each input
  - Attack 8: Force invalid outputs to be generated
  - Attack 9: Force properties of an output to change
  - Attack 10: Force the screen to refresh.

# *How to Break Software attacks*

## **Testing from the user interface: Data and computation**

- **Exploring stored data**

- Attack 11: Apply inputs using a variety of initial conditions
- Attack 12: Force a data structure to store too many or too few values
- Attack 13: Investigate alternate ways to modify internal data constraints

- **Exploring computation and feature interaction**

- Attack 14: Experiment with invalid operand and operator combinations
- Attack 15: Force a function to call itself recursively
- Attack 16: Force computation results to be too large or too small
- Attack 17: Find features that share data or interact poorly

# *How to Break Software attacks*

## **System interface attacks**

- **Testing from the file system interface: Media-based attacks**
  - Attack 1: Fill the file system to its capacity
  - Attack 2: Force the media to be busy or unavailable
  - Attack 3: Damage the media
- **Testing from the file system interface: File-based attacks**
  - Attack 4: Assign an invalid file name
  - Attack 5: Vary file access permissions
  - Attack 6: Vary or corrupt file contents



# Attack Example: Interference Testing

- We're often looking at asynchronous events here. One task is underway, and we do something to interfere with it.
- In many cases, the critical event is extremely time sensitive. For example:
  - An event reaches a process just as, just before, or just after it is timing out or just as (before / during / after) another process that communicates with it will time out listening to this process for a response. (“Just as?” —if special code is executed in order to accomplish the handling of the timeout, “just as” means during execution of that code)
  - An event reaches a process just as, just before, or just after it is servicing some other event.
  - An event reaches a process just as, just before, or just after a resource needed to accomplish servicing the event becomes available or unavailable.
- ***So let's look for ways to manipulate the time relationships among potentially relevant events***



# Attack Example: Interference Testing

## **Interference testing: Generate interrupts**

- From a device related to the task (e.g. pull out a paper tray, perhaps one that isn't in use while the printer is printing)
- From a device unrelated to the task (e.g. move the mouse and click while the printer is printing)
- From a software event (e.g. set another program's (or this program's) time-reminder to go off during the task under test)

## **Change something that this task depends on**

- Swap out a floppy
- Change the contents of a file that this program is reading
- Change the printer that the program will print to (without signaling a new driver)
- Change the video resolution

# Attack Example: Interference Testing

## **Interference testing: Cancel**

- Cancel the task (at different points during its completion)
- Cancel some other task while this task is running
  - a task in communication with this task (the core task being studied)
  - a task that will eventually have to complete as a prerequisite to completion of this task
  - a task totally unrelated to this task

## **Interference testing: Pause**

- Find some way to create a temporary interruption in the task.
- Pause the task
  - for a short time
  - for a long time (long enough for a timeout, if one will arise)
- Put the printer on local
- Put a database under use by a competing program, lock a record so that it can't be accessed — yet.

# Attack Example: Interference Testing

## Interference testing: Swap (out of memory)

- Swap the process out of memory while it's running (e.g. change focus to another application; keep loading or adding applications until the application under test is paged to disk.
  - Leave it swapped out for 10 minutes or whatever the timeout period is. Does it come back? What is its state? What is the state of processes that are supposed to interact with it?
  - Leave it swapped out *much* longer than the timeout period. Can you get it to the point where it is supposed to time out, then send a message that is supposed to be received by the swapped-out process, then time out on the time allocated for the message? What are the resulting state of this process and the one(s) that tried to communicate with it?
- Swap a related process out of memory while the process under test is running.

# Attack Example: Interference Testing

## Interference testing: Compete

- *Compete for a device (such as a printer)*
  - put device in use, then try to use it from software under test
  - start using device, then use it from other software
  - If there is a priority system for device access, use software that has higher, same and lower priority access to the device before and during attempted use by software under test
- *Compete for processor attention*
  - some other process generates an interrupt (e.g. ring into the modem, or a time-alarm in your contact manager)
  - try to do something during heavy disk access by another process
- *Send this process another job while one is underway*

# Exploration & Learning

- Explorers use a variety of tactics to learn about the product and its context (market, environment, etc.)
- There are enormous individual and contextual differences
- Here are a few examples.
  - Early coverage heuristics
  - Active reading of reference materials
  - Attacks
  - **Failure mode lists**
  - Project risk lists
  - Scenario development
  - Models



## Failure mode lists / Risk catalogs / Bug taxonomies

- A **failure mode** is, essentially, a way the program could fail.
- Example: **Portion of risk catalog for installer products:**
  - Wrong files installed
    - temporary files not cleaned up
    - old files not cleaned up after upgrade
    - unneeded file installed
    - needed file not installed
    - correct file installed in the wrong place
  - Files clobbered
    - older file replaces newer file
    - user data file clobbered during upgrade
  - Other apps clobbered
    - file shared with another product is modified
    - file belonging to another product is deleted



# Failure modes can guide testing

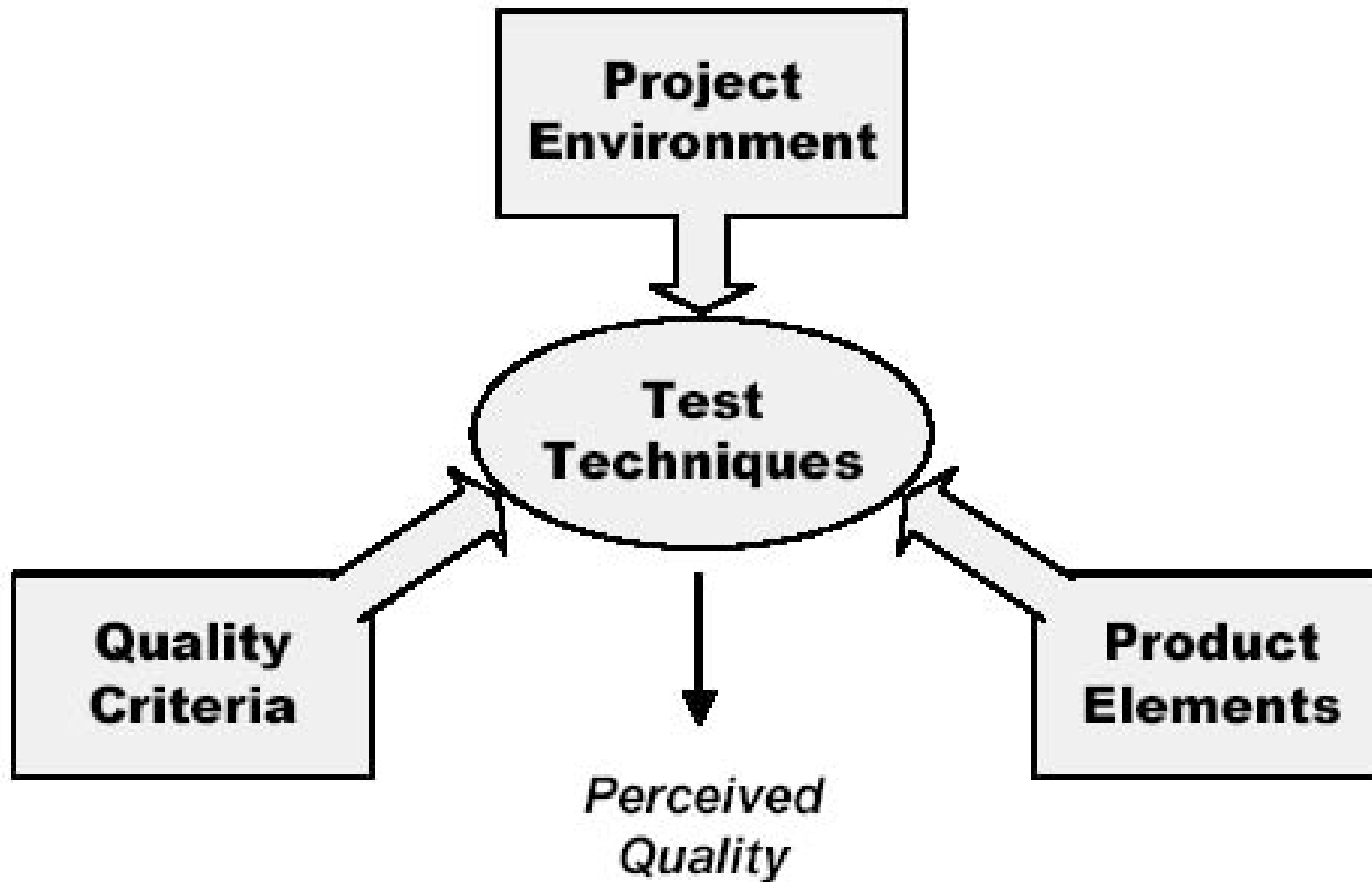
*Testing Computer Software* listed 480 common bugs. We used the list for:

- **Test idea generation**
  - 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.
- **Test plan auditing**
  - Pick categories to sample from
  - From each category, find a few potential defects in the list
  - For each potential defect, ask whether the software under test could have this defect
  - If it is theoretically possible that the program could have the defect, ask whether the test plan could find the bug if it was there.
- **Getting unstuck**
  - Look for classes of problem outside of your usual box
- **Training new staff**
  - Expose them to what can go wrong, challenge them to design tests that could trigger those failures

# Build your own catalog of failure modes

- 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.
- There are plenty of sources to check for common failures in the common platforms
  - [www.bugnet.com](http://www.bugnet.com)
  - [www.cnet.com](http://www.cnet.com)
  - trade press gossip about bugs
  - links from [www.winfiles.com](http://www.winfiles.com)
  - various mailing lists

# Building failure mode lists: Bach's heuristic test strategy model

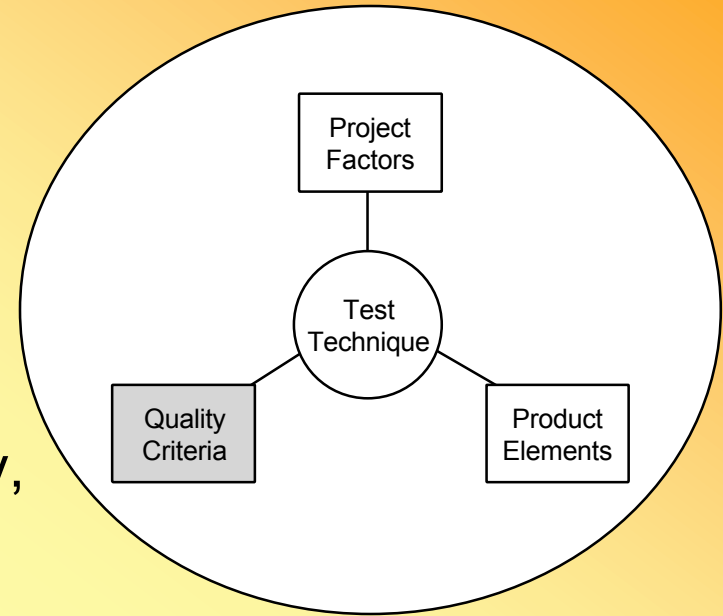


# Risk-based testing: Failure modes

- Imagine how the product can fail, then design/use tests that can expose this type of failure. We analyze these from three angles:
  - **Quality attribute failures:** What quality attributes (e.g. accessibility, usability, maintainability) do we most care about, or for each attribute, how could we expose inadequacies in the product?
  - **Product element (or, component) failures:** What are the parts of the product and, for each part, how could it fail?
    - **Operational failures:** How do things go wrong when we use the product to do real things? *Because the issues overlap so much, we currently fold these into the product element and quality attribute analyses.*
  - **Common programming errors:** What types of mistakes are common, and for each type of error, is there a specific technique -- *an attack* -- optimized for exposing this kind of mistake?

# Quality criteria

- Quality criteria are high-level oracles. Use them to determine or argue that the product has problems. Quality criteria are multidimensional, often in conflict with each other.
- Each quality criterion is a risk category, such as
  - “the risk of inaccessibility.”
- Failure mode listing:
  - Build a list of examples of each category.
  - Look for bugs similar to the examples.



# Quality criteria categories: Operational criteria

- **Capability**. Can it perform the required functions?
- **Reliability**. Will it work well and resist failure in all required situations?
  - **Error handling**: product resists failure in response to errors, is graceful when it does fail, and recovers readily.
  - **Data Integrity**: data in the system is protected from loss or corruption.
  - **Security**: product is protected from unauthorized use.
  - **Safety**: product will not fail in such a way as to harm life or property.



# Quality criteria categories: Operational criteria

- **Usability.** How easy is it for a real user to use the product?
  - **Learnability:** operation of the product can be rapidly mastered by the intended user.
  - **Operability:** product can be operated with minimum effort and fuss.
  - **Throughput:** how quickly the user can do the complete task.
  - **Accessibility:** product is usable in the face of the user's limitations or disabilities
- **Performance.** How speedy and responsive is it?
- **Concurrency:** appropriately handles multiple parallel tasks and behaves well when running in parallel with other processes.
- **Scalability:** appropriately handles increases in number of users, tasks, or attached resources.

# Quality criteria categories: Operational criteria

- ***Installability and Uninstallability***. How easily can it be installed onto (and uninstalled from) its target platform?
- ***Compatibility***. How well does it work with external components & configurations?
  - ***Application Compatibility***: product works in conjunction with other software products.
  - ***Operating System Compatibility***: product works with a particular operating system.
  - ***Hardware Compatibility***: product works with particular hardware components and configurations.
  - ***Backward Compatibility***: the product works with earlier versions of itself.
  - ***Resource Usage***: the product doesn't unnecessarily hog memory, storage, or other system resources.

# Quality criteria categories: Development criteria

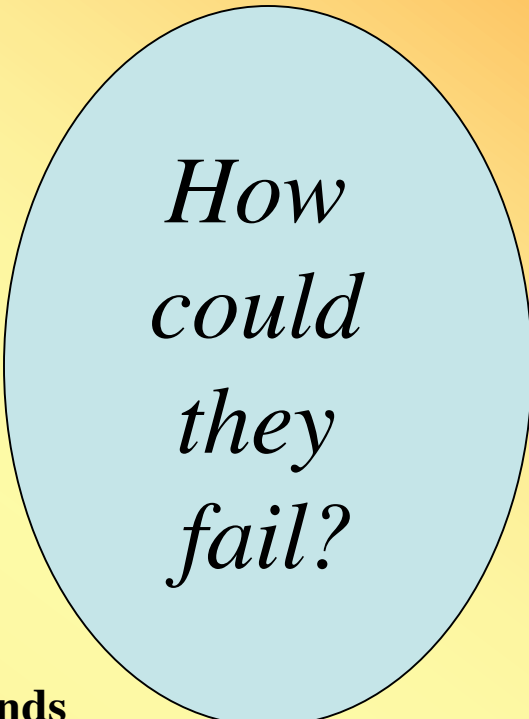
- **Supportability.** How economical will it be to provide support to users of the product?
- **Testability.** How effectively can the product be tested?
- **Maintainability.** How economical is it to build, fix or enhance the product?
- **Portability.** How economical will it be to port or reuse the technology elsewhere?
- **Localizability.** How economical will it be to publish the product in another language?

# Quality criteria categories: Development criteria

- **Conformance to Standards.** How well it meets standardized requirements, such as:
  - **Coding Standards.** Agreements among the programming staff or specified by the customer.
  - **Regulatory Standards.** Attributes of the program or development process made compulsory by legal authorities.
  - **Industry Standards.** Includes widely accepted guidelines (e.g. user interface conventions common to a particular UI) and formally adopted standards (e.g IEEE standards).

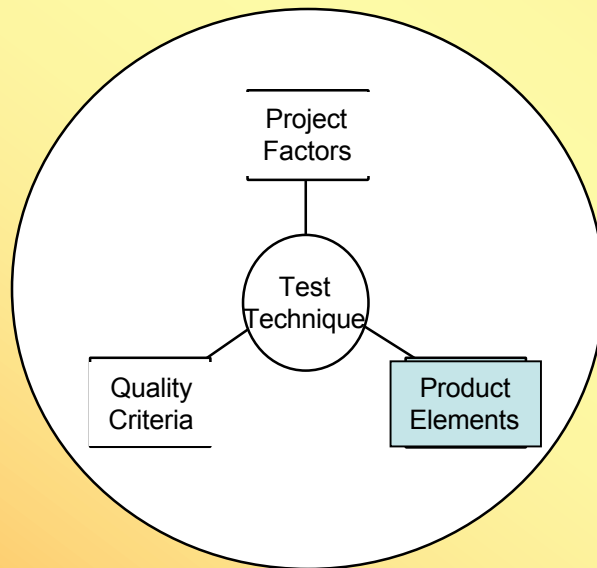
# Building failure mode lists from product elements: Shopping cart example

- Think in terms of the components of your product
  - **Structures: Everything that comprises the logical or physical product**
    - Database server
    - Cache server
  - **Functions: Everything the product does**
    - Calculation
    - Navigation
    - Memory management
    - Error handling
  - **Data: Everything the product processes**
    - Human error (retailer)
    - Human error (customer)
  - **Operations: How the product will be used**
    - Upgrade
    - Order processing
  - **Platforms: Everything on which the product depends**
    - » Adapted from Giri Vijayaraghavan's Master's thesis.



*How  
could  
they  
fail?*

## 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, graphics, movies, etc.
  - ***Ephemera and collaterals***: anything beyond software and hardware, such as paper documents, web links and content, packaging, license agreements, etc.

# Product elements

- **Functions: *Everything 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 fulfil core requirements.
  - ***Multimedia:*** self-executing or automatically executing sounds, bitmaps, movies embedded in the product.
  - ***Error Handling:*** functions that detect and recover from errors, including all error messages.
  - ***Interactions:*** interactions or interfaces between functions within the product.
  - ***Testability:*** functions provided to help test the product, such as diagnostics, log files, asserts, test menus, etc.

# Product elements

- **Data: *Everything 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 prefabricated 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***: any relationship between data and time, such as the number of keystrokes per second, date stamps on files, or synchronization of distributed systems.

# 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.

# Product elements

- **Operations: *How the product will be used***
  - *Users*. attributes of the various kinds of users.
  - *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.

# Analyzing Mobile Applications

- Ajay Jha (a Masters' candidate at in Kaner's lab at Florida Tech) is applying Bach's analytical structure to mobile applications.
- The goal is to develop an idea generator for testing these applications:
  - What is special about testing applications that run on PDAs, cell phones, and other network-connected small devices
- His current draft analysis is attached
  
- (By the way, Ajay is looking for an internship, to gain industry experience with these issues. Have you any openings?)



# 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 for 101 examples of coverage “measures” and Measurement of the Extent of Testing for a broader discussion of measurement theory and coverage, both at [www.kaner.com](http://www.kaner.com)*

# Exploration & Learning

- Explorers use a variety of tactics to learn about the product and its context (market, environment, etc.)
- There are enormous individual and contextual differences
- Here are a few examples.
  - Early coverage heuristics
  - Active reading of reference materials
  - Attacks
  - Failure mode lists
  - **Project risk lists**
  - Scenario development
  - Models

# Classic, project-level risk analysis

Project-level risk analyses usually consider risks factors that can make the project as a whole fail, and how to manage those risks.

The screenshot shows a presentation slide within an Acrobat Reader window. The slide is divided into two main sections: 'Categories of Risk Sources' on the left and 'Project Consequences' on the right, connected by a large blue arrow pointing from left to right. The 'Categories of Risk Sources' section lists 13 items, and the 'Project Consequences' section lists 10 items. The Acrobat Reader window title is 'Acrobat Reader - [tutorial from sei 2.pdf]'. The status bar at the bottom of the window shows 'Page 12 of 53', '146%' zoom, and '9.54 x 7.28 in' dimensions. The TeraQuest logo is in the bottom left, and 'SEPG Risk Workshop © 1998 TeraQuest' is in the bottom right of the slide area.

Categories of Risk Sources	Project Consequences
■ Mission and goals	■ Cost overruns
■ Decision drivers	■ Schedule slips
■ Organization management	■ Inadequate functionality
■ Customer / end user	■ Canceled projects
■ Budget / cost	■ Sudden personnel changes
■ Schedule	■ Customer dissatisfaction
■ Project characteristics	■ Loss of company image
■ Development process	■ Demoralized staff
■ Development environment	■ Poor product performance
■ Personnel	■ Legal proceedings
■ Operational environment	
■ New technology	

# Project-level risk analysis

## Project risk management involves

- Identification of the different risks to the project (issues that might cause the project to fail or to fall behind schedule or to cost too much or to dissatisfy customers or other stakeholders)
- Analysis of the potential costs associated with each risk
- Development of plans and actions to reduce the likelihood of the risk or the magnitude of the harm
- Continuous assessment or monitoring of the risks (or the actions taken to manage them)
- Useful material available free at <http://seir.sei.cmu.edu>
- <http://www.coyotevalley.com> (Brian Lawrence)

The problem for our purposes is that this level of analysis doesn't give us much guidance as to how to test.

# Risk heuristics: Where to look for errors

Sometimes risks associated with the project as a whole or with the staff or management of the project can guide our testing.

- **New things:** newer features may fail.
- **New technology:** new concepts lead to new mistakes.
- **Learning Curve:** mistakes due to ignorance.
- **Changed things:** changes may break old code.
- **Late change:** rushed decisions, rushed or demoralized staff lead to mistakes.
- **Rushed work:** some tasks or projects are chronically underfunded and all aspects of work quality suffer.
- **Tired programmers:** long overtime over several weeks or months yields inefficiencies and errors

» Adapted from James Bach's lecture notes



# Risk heuristics: Where to look for errors

- **Other staff issues:** alcoholic, mother died, two programmers who won't talk to each other (neither will their code)...
- **Just slipping it in:** pet feature not on plan may interact badly with other code.
- **N.I.H.:** external components can cause problems.
- **N.I.B.:** (not in budget) Unbudgeted tasks may be done shoddily.
- **Ambiguity:** ambiguous descriptions (in specs or other docs) can lead to incorrect or conflicting implementations.

» Adapted from James Bach's lecture notes



# Risk heuristics: Where to look for errors

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

**Unknown requirements:** requirements surface throughout development. Failure to meet a legitimate requirement is a failure of quality for that stakeholder.

**Evolving requirements:** people realize what they want as the product develops. Adhering to a start-of-the-project requirements list may meet the contract but yield a failed product. (check out <http://www.agilealliance.org/>)

**Complexity:** complex code may be buggy.

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

» Adapted from James Bach's lecture notes

# Risk heuristics: Where to look for errors

- **Dependencies:** failures may trigger other failures.
- **Untestability:** risk of slow, inefficient testing.
- **Little unit testing:** programmers find and fix most of their own bugs. Shortcutting here is a risk.
- **Little system testing so far:** untested software may fail.
- **Previous reliance on narrow testing strategies:** (e.g. regression, function tests), can yield a backlog of errors surviving across versions.
- **Weak testing tools:** if tools don't exist to help identify / isolate a class of error (e.g. wild pointers), the error is more likely to survive to testing and beyond.

» Adapted from James Bach's lecture notes

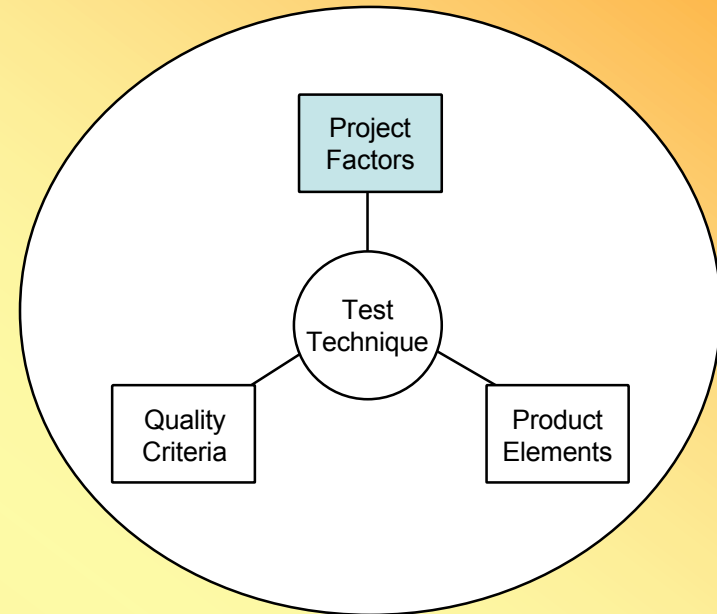
## Risk heuristics: Where to look for errors

- **Unfixability:** risk of not being able to fix a bug.
- **Language-typical errors:** such as wild pointers in C. See
  - Bruce Webster, *Pitfalls of Object-Oriented Development*
  - Michael Daconta et al. *Java Pitfalls*
- **Criticality:** severity of failure of very important features.
- **Popularity:** likelihood or consequence if much used features fail.
- **Market:** severity of failure of key differentiating features.
- **Bad publicity:** a bug may appear in PC Week.
- **Liability:** being sued.

» Adapted from James Bach's lecture notes

# Heuristic test strategy model: Project environment factors

- **Customers.** *Anyone who is a client of the test project.*
- **Information.** *Information about the product or project that is needed for testing.*
- **Team.** *Anyone who will perform or support testing.*
- **Equipment & Tools.** *Hardware, software, or documents required to administer testing.*
- **Schedules.** *The sequence, duration, and synchronization of events.*
- **Test Items.** *The product to be tested.*
- **Deliverables.** *The observable products of the test project.*
- **Logistics and Budget.**



**These aspects of the environment constrain and enable the testing project**

# Project environment factors

- **Customers.** Anyone who is a client of the test project.
  - Do you know who your customers are? Whose opinions matter? Who benefits or suffers from the work you do?
  - Do you have contact and communication with your customers?
  - Maybe your customers have strong ideas about what tests you should create and run.
  - Maybe they have conflicting expectations. You may have to help identify and resolve those.
  - Maybe they can help you test, in some way.

# Project environment factors

- **Information.** Information about the product or project that is needed for testing.
  - Do you have all the information that you need in order to test reasonably well?
  - Do you need to familiarize yourself with the product more, before you will know how to test it?
  - Is your information current? How are you apprised of new or changing information?



# Project environment factors

- **Team.** Anyone who will perform or support testing.
  - Do you know who will be testing?
  - Are there people not on the “test team” who might be able to help? People who’ve tested similar products before and might have advice? Writers? Users? Programmers?
  - Do you have enough people with the right skills to fulfill a reasonable test strategy?
  - Does the team have special skills or motivation to use particular test techniques?
  - Is any training needed? Is any available?
  - Extent to which they are focused or are multi-tasking?
  - Organization: collaboration & coordination of the staff?
  - Is there an independent test lab?

# Project environment factors

- **Equipment & Tools.** Hardware, software, or documents required to administer testing.
  - *Hardware:* Do we have all the equipment you need to execute the tests? Is it set up and ready to go?
  - *Automation:* Are any test automation tools needed? Are they available?
  - *Probes / diagnostics:* Which tools needed to aid observation of the product under test?
  - *Matrices & Checklists:* Are any documents needed to track or record the progress of testing? Do any exist?

# Project environment factors

- **Schedules.** The sequence, duration, and synchronization of events.
  - *Testing:* How much time do you have? Are there tests better to create later than to create them now?
  - *Development:* When will builds be available for testing, features added, code frozen, etc.?
  - *Documentation:* When will the user documentation be available for review?
  - *Hardware:* When will the hardware you need to test with (more generally, the 3rd party materials you need) be available and set up?

# Project environment factors

- **Test Items.** The product to be tested.
  - *Availability:* Do you have the product to test?
  - *Volatility:* Is the product constantly changing? What will be the need for retesting?
  - *Testability:* Is the product functional and reliable enough that you can effectively test it?

# Project environment factors

- **Deliverables.** The observable products of the test project.
  - *Content:* What sort of reports will you have to make? Will you share your working notes, or just the end results?
  - *Purpose:* Are your deliverables provided as part of the product? Does anyone else have to run your tests?
  - *Standards:* Is there a particular test documentation standard you're supposed to follow?
  - *Media:* How will you record and communicate your reports?

# Project environment factors

- **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.?



# Exploration & Learning

- Explorers use a variety of tactics to learn about the product and its context (market, environment, etc.)
- There are enormous individual and contextual differences
- Here are a few examples.
  - Early coverage heuristics
  - Active reading of reference materials
  - Attacks
  - Failure mode lists
  - Project risk lists
  - **Scenario development**
  - Models

# Scenario testing

- We can use any technique in the course of exploratory testing. As we get past the introductory tests, we need strategies for gaining a deeper knowledge of the program and applying that to tests. Scenario testing is a classic method for this.
- The ideal scenario has several characteristics:
  - The test is *based on a story* about how the program is used, including information about the motivations of the people involved.
  - The story is *motivating*. A stakeholder with influence would push to fix a program that failed this test.
  - The story is *credible*. It not only *could* happen in the real world; stakeholders would believe that something like it probably *will* happen.
  - The story involves a *complex use* of the program *or a complex environment or a complex set of data*.
  - The test results are *easy to evaluate*. This is valuable for all tests, but is especially important for scenarios because they are complex.

# Why use scenario tests?

- Learn the product
- Connect testing to documented requirements
- Expose failures to deliver desired benefits
- Explore expert use of the program
- Make a bug report more motivating
- Bring requirements-related issues to the surface, which might involve reopening old requirements discussions (with new data) or surfacing not-yet-identified requirements.

# Scenarios

- Designing scenario tests is much like doing a requirements analysis, but is not requirements analysis. They rely on similar information but use it differently.
  - The requirements analyst tries to foster agreement about the system to be built. The tester exploits disagreements to predict problems with the system.
  - The tester doesn't have to reach conclusions or make recommendations about how the product should work. Her task is to expose credible concerns to the stakeholders.
  - The tester doesn't have to make the product design tradeoffs. She exposes the consequences of those tradeoffs, especially unanticipated or more serious consequences than expected.
  - The tester doesn't have to respect prior agreements. (Caution: testers who belabor the wrong issues lose credibility.)
  - The scenario tester's work need not be exhaustive, just useful.

# Scenarios

Some ways to trigger thinking about scenarios:

- **Benefits-driven:** People want to achieve X. How will they do it, for the following X's?
- **Sequence-driven:** People (or the system) typically do task X in an order. What are the most common orders (sequences) of subtasks in achieving X?
- **Transaction-driven:** We are trying to complete a specific transaction, such as opening a bank account or sending a message. What are all the steps, data items, outputs and displays, etc.?
- **Get use ideas from competing product:** Their docs, advertisements, help, etc., all suggest best or most interesting uses of their products. How would our product do these things?
- **Competitor's output driven:** Hey, look at these cool documents they can make. Look (think of Netscape's superb handling of often screwy HTML code) at how well they display things. How do we do with these?
- **Customer's forms driven:** Here are the forms the customer produces in her business. How can we work with (read, fill out, display, verify, whatever) them?



# Twelve ways to create good scenarios

1. Write life histories for objects in the system.
2. List possible users, analyze their interests and objectives.
3. Consider disfavored users: how do they want to abuse your system?
4. List system events. How does the system handle them?
5. List special events. What accommodations does the system make for these?
6. List benefits and create end-to-end tasks to check them.
7. Interview users about famous challenges and failures of the old system.
8. Work alongside users to see how they work and what they do.
9. Read about what systems like this are supposed to do.
10. Study complaints about the predecessor to this system or its competitors.
11. Create a mock business. Treat it as real and process its data.
12. Try converting real-life data from a competing or predecessor application.



# Exploration & Learning

- Explorers use a variety of tactics to learn about the product and its context (market, environment, etc.)
- There are enormous individual and contextual differences
- Here are a few examples.
  - Early coverage heuristics
  - Active reading of reference materials
  - Attacks
  - Failure mode lists
  - Project risk lists
  - Scenario development
  - **Models**

# Models

- If you can develop a model of the product:
  - you can test the model as you develop it
  - you can draw implications from the model
- For the explorer, the modeling effort is successful if it leads to interesting tests
  - in a reasonable time frame, or
  - that would be too hard to think of in other ways
- Examples of models:
  - architecture diagram
  - state-based
  - dataflow

# Example: Exploring Architecture Diagrams

- **Work from a high level design (map) of the system**
  - pay primary attention to interfaces between components or groups of components. We're looking for cracks that things might have slipped through
  - what can we do to screw things up as we trace the flow of data or the progress of a task through the system?
- **You can build the map in an architectural walkthrough**
  - Invite several programmers and testers to a meeting. Present the programmers with use cases and have them draw a diagram showing the main components and the communication among them. For a while, the diagram will change significantly with each example. After a few hours, it will stabilize.
  - Take a picture of the diagram, blow it up, laminate it, and you can use dry erase markers to sketch your current focus.
  - Planning of testing from this diagram is often done jointly by several testers who understand different parts of the system.

# Key challenges of exploratory testing

- Software testing poses several core challenges to the skilled practitioner / manager. The same challenges apply to exploratory testers:
  - **Learning** (How do we get to know the program?)
  - **Visibility** (How to see below the surface?)
  - **Control** (How to set internal data values?)
  - **Risk / selection** (Which are the best tests to run?)
  - **Execution** (What's the most efficient way to run the tests?)
  - **Logistics** (What environment is needed to support test execution?)
  - **The oracle problem** (How do we tell if a test result is correct?)
  - **Reporting** (How can we replicate a failure and report it effectively?)
  - **Documentation** (What test documentation do we need?)
  - **Measurement** (What metrics are appropriate?)
  - **Stopping** (How to decide when to stop testing?)