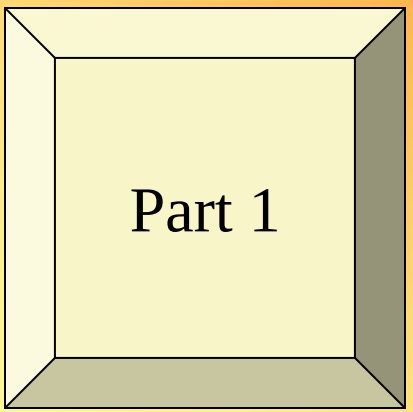
Black Box Software Testing

Fall 2005

RISK-BASED 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.

Introduction to risk-based testing

Risk: The possibility of suffering harm or loss

- In software testing, we think of risk on three dimensions:
  - *A way the program could fail*
  - *How likely it is that the program could fail in that way*
  - *What the consequences of that failure could be*

For testing purposes, the most important is:

- *A way the program could fail*

For project management purposes,

- *How likely*
- *What consequences*

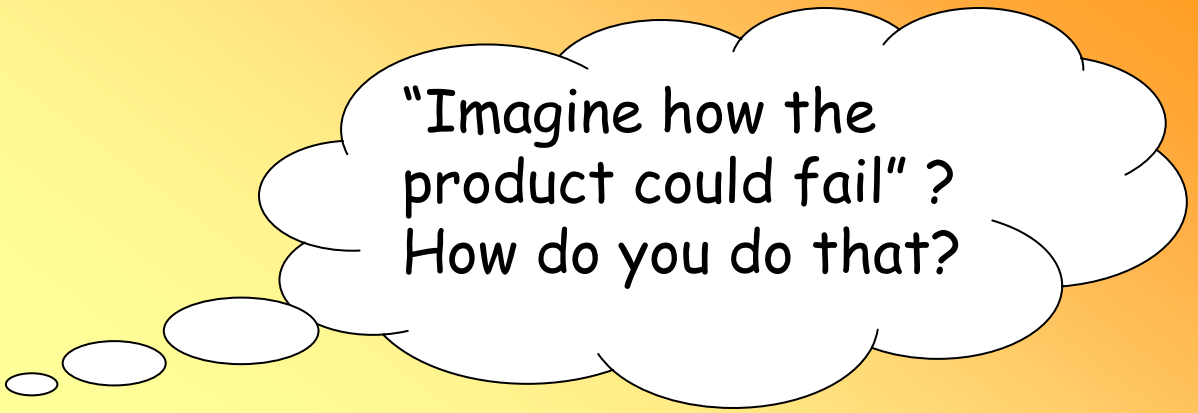
A way the program could fail

- The essence of risk-based testing is this:
  - Imagine how the product could fail
  - Design tests to expose these (potential) failures

Risk-based domain testing

- For example, in the risk-based approach to domain testing:
  - We identify a space (probably associated with one or a few variables) that has far too many potential tests.
    - The domain testing objective is to partition this space and pick a few representative tests, making the testing manageable.
  - We then imagine risks (problems the program *could have* that involve the space of interest)
  - For each risk:
    - Some tests could expose the problem, others could not
    - The error-revealing tests make up one or more equivalence classes.
      - Identify the class(es)
      - Pick one or a few best representatives of each class

Just one problem



Just one problem

"Imagine how the product could fail" ?
How do you do that?

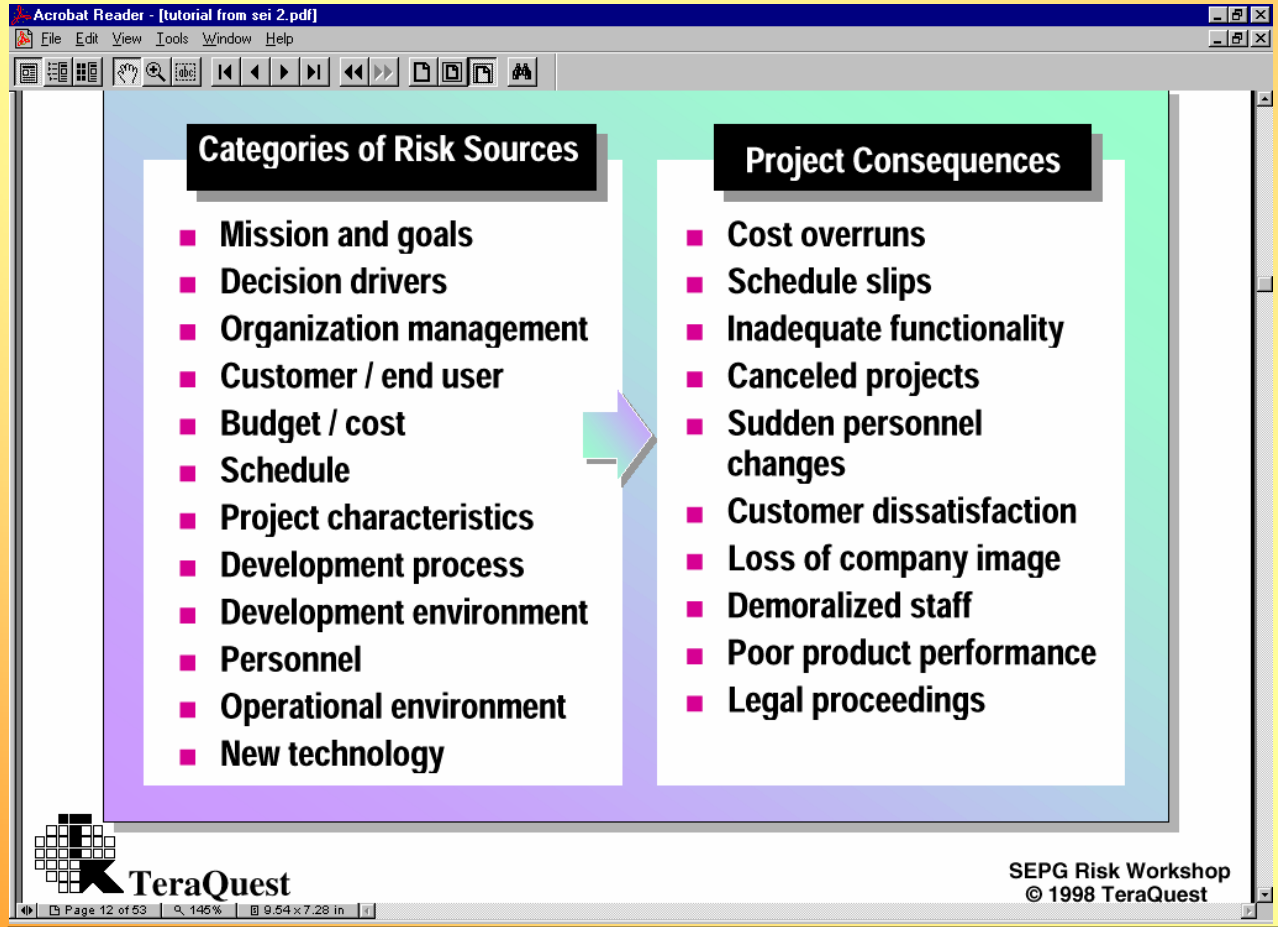
We'll consider three classes of heuristics:

- Recognize common project warning signs (and test things associated with the risky aspects of the project).
- Apply failure mode and effects analysis to (many or all) elements of the product and to the product's key quality criteria.
- Apply common techniques (*quicktests* or *attacks*) to take advantage of common errors

We call these heuristics because they fallible but useful guides. You have to exercise your own judgment about which to use when.

Project-Level Risk Factors

Classic, project-level risk analysis



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

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.

Project risk heuristics: Where to look for errors

- **New things**: less likely to have revealed its bugs yet.
- **New technology:** same as new code, plus the risks of unanticipated problems.
- **Learning curve:** people make more mistakes while learning.
- **Changed things**: same as new things, but changes can also break old code.
- **Poor control**: without SCM, files can be overridden or lost.
- **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.

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

Project risk heuristics: Where to look for errors

- **Fatigue:** tired people make mistakes.
- **Distributed team:** a far flung team communicates less.
- **Other staff issues:** alcoholic, mother died, two programmers who won't talk to each other (neither will their code)…
- **Surprise features:** features not carefully planned may have unanticipated effects on other features.
- **Third-party code:** external components may be much less well understood than local code, and much harder to get fixed.
- **Unbudgeted:** unbudgeted tasks may be done shoddily.

Project risk heuristics: Where to look for errors

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

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

Mysterious silence: when something interesting or important is not described or documented, it may have not been thought through, or the designer may be hiding its problems.

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.

Project risk heuristics: Where to look for errors

Buggy: anything known to have lots of problems has more.

Recent failure: anything with a recent history of problems.

Upstream dependency: may cause problems in the rest of the system.

Downstream dependency: sensitive to problems in the rest of the system.

Distributed: anything spread out in time or space, that must work as a unit.

Open-ended: any function or data that appears unlimited.

Complex: what's hard to understand is hard to get right.

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

Project risk heuristics: Where to look for errors

- **Little system testing**: untested software will fail.
- **Little unit testing:** programmers normally find and fix most of their own bugs.
- **Previous reliance on narrow testing strategies:** can yield a many-version backlog of errors not exposed by those techniques.
- **Weak test 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.
- **Unfixable:** bugs that survived because, when they were first reported, no one knew how to fix them in the time available.
- **Untestable:** anything that requires slow, difficult or inefficient testing is probably undertested.

Project risk heuristics: Where to look for errors

- **Publicity**: anywhere failure will lead to bad publicity.
- **Liability**: anywhere that failure would justify a lawsuit.
- **Critical**: anything whose failure could cause substantial damage.
- **Precise**: anything that must meet its requirements exactly.
- **Easy to misuse**: anything that requires special care or training to use properly.
- **Popular**: anything that will be used a lot, or by a lot of people.
- **Strategic**: anything that has special importance to your business.
- **VIP**: anything used by particularly important people.
- **Visible**: anywhere failure will be obvious and upset users.
- **Invisible**: anywhere failure will be hidden and remain undetected until a serious failure results.

Failure Modes

Failure Mode & Effects Analysis (FMEA)

Bug catalogs can guide testing

Testing Computer Software listed almost 500 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

Failure mode lists / Risk catalogs / Bug taxonomies

- A *failure mode* is, essentially, a way that 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 mode & effects analysis

- Widely used for safety analysis of goods.
- Consider the product in terms of its components. For each component
  - Imagine the ways it could fail. For each potential failure (each failure mode), ask questions:
    - What would that failure look like?
    - How would you detect that failure?
    - How expensive would it be to search for that failure?
    - Who would be impacted by that failure?
    - How much variation would there be in the effect of the failure?
    - How serious (on average) would that failure be?
    - How expensive would it be to fix the underlying cause?
  - On the basis of the analysis, decide whether it is cost effective to search for this potential failure

Building a failure mode catalog

- Giri Vijayaraghavan and Ajay Jha (thesis in progress) followed similar approaches in developing their catalogs:
  - Identify components
    - They used the Satisfice Test Strategy Model as a starting point. We'll study it soon, in another lecture.
    - Imagine ways the program could fail (in this component).
      - They used magazines, web discussions, some corporations' bug databases, interviews with people who had tested their class of products, and so on, to guide their imagination.
    - Imagine failures involving interactions among components
  - They did the same thing for quality attributes (see next section).
- These catalogs are not orthogonal. They help generate test ideas, but are not suited for classifying test ideas.

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**

How could they fail?

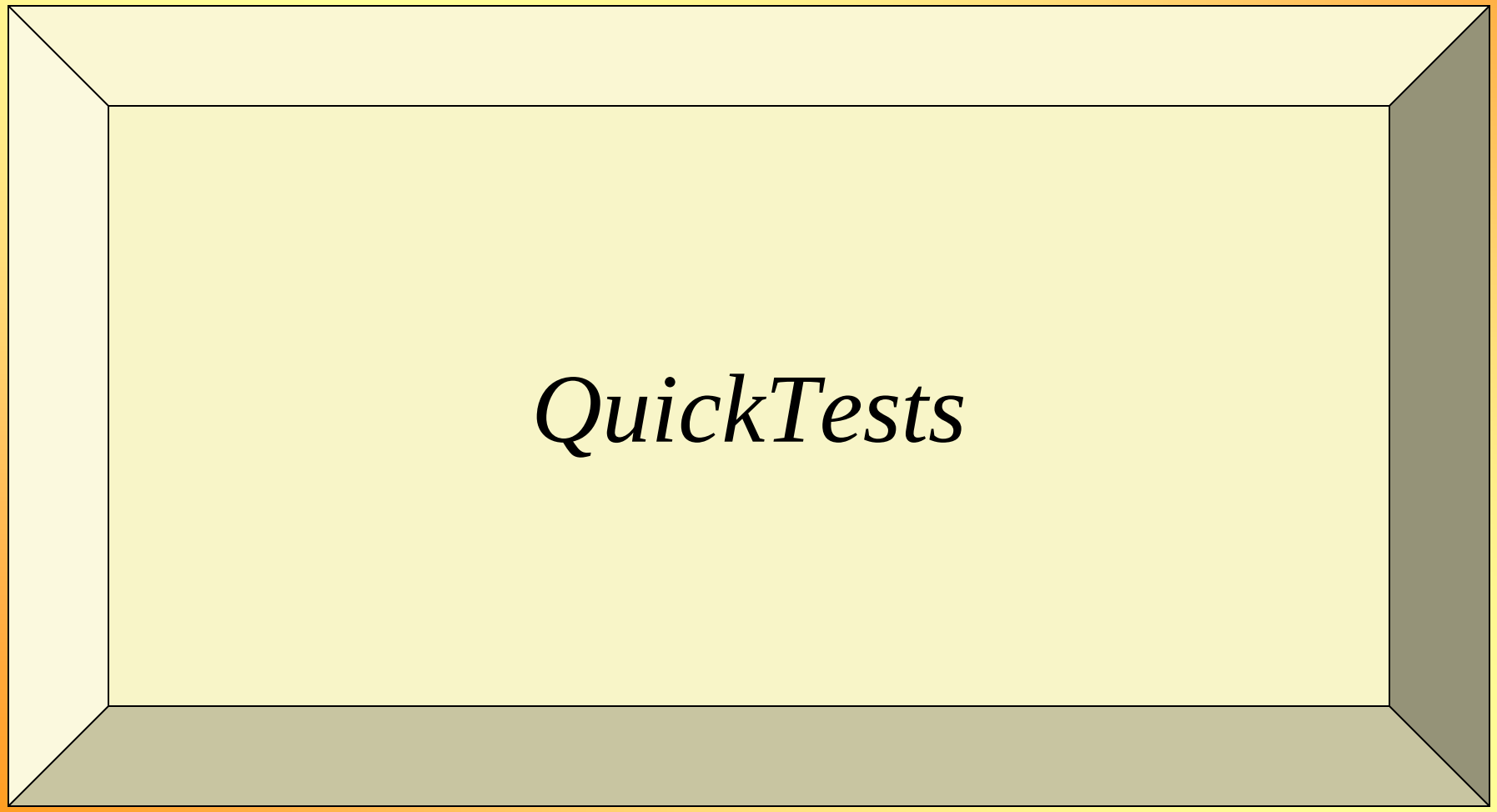
» Adapted from Giri Vijayaraghavan's Master's thesis.

Extending FMEA to Quality Attributes

FMEA & quality attributes

- In FMEA, we list a bunch of things we could test, and then figure out how they might fail.
- Quality attributes cut across the components:
  - **Usability**
    - Easy to learn
    - Reasonable number of steps
    - Accessible to someone with a disability
      - Auditory
      - Visual
        - » *Imagine evaluating every product element in terms of accessibility to someone with a visual impairment.*

Risk-based testing



QuickTests?

- A ***quicktest*** is a cheap test that has some value but requires little preparation, knowledge, or time to perform.
  - Participants at the 7th Los Altos Workshop on Software Testing (Exploratory Testing, 1999) pulled together a collection of these.
  - James Whittaker published another collection in *How to Break Software*.
  - Elisabeth Hendrickson teaches courses on bug hunting techniques and tools, many of which are quicktests or tools that support them.
  -

A Classic QuickTest: The Shoe Test

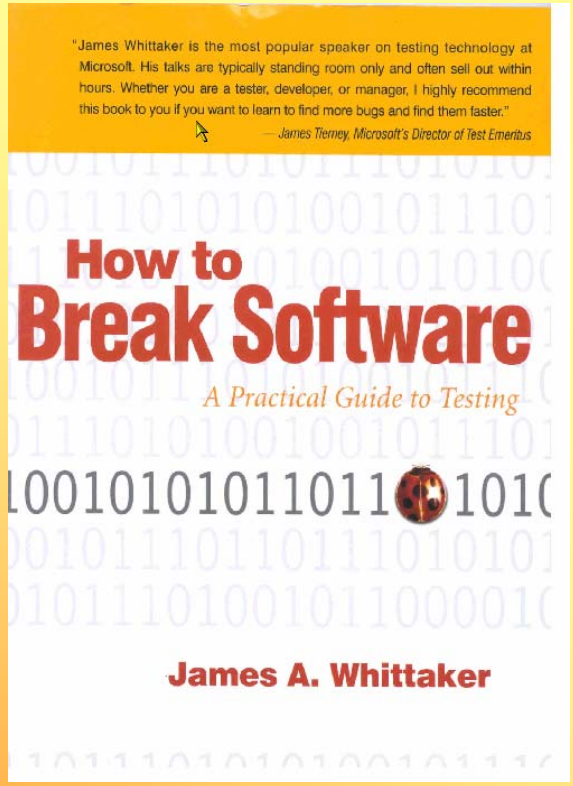
- Find an input field, move the cursor to it, put your shoe on the keyboard, and go to lunch.
- Basically, you're using the auto-repeat on the keyboard for a cheap stress test.
  - *Tests like this often overflow input buffers.*
- In Bach's favorite variant, he finds a dialog box so constructed that pressing a key leads to, say, another dialog box (perhaps an error message) that also has a button connected to the same key that returns to the first dialog box.
  - *This will expose some types of long-sequence errors (stack overflows, memory leaks, etc.)*

Another Classic Example of a QuickTest

- Traditional boundary testing
  - All you need is the variable, and its possible values.
  - You need very little information about the meaning of the variable (why people assign values to it, what it interacts with).
  - You test at boundaries because miscoding of boundaries is a common error.
- Note the foundation of this test.

> There is a programming error so common that it's worth building a test technique optimized to find errors of that type.

"Attacks" to expose common coding errors



- Jorgensen & Whittaker pulled together a collection of common coding errors, many of them involving insufficiently or incorrectly constrained variables.
- They created (or identified common) *attacks* to test for these.
- An *attack* is a stereotyped class of tests, optimized around a specific type of error.

Think back to boundary testing:

- ***Boundary testing for numeric input fields is an example of an attack. The error is mis-specification (or mis-typing) of the upper or lower bound of the numeric input field.***

"Attacks" to expose common coding errors

- 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.

"Attacks" to expose common coding errors

- *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
    - » From Whittaker, *How to Break Software*

"Attacks" to expose common coding errors

- *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.

» From Whittaker, *How to Break Software*

"Attacks" to expose common coding errors

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

» From Whittaker, *How to Break Software*

"Attacks" to expose common coding errors

Testing from the user interface: Data and computation

- 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

» From Whittaker, *How to Break Software*

"Attacks" to expose common coding errors

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
    - » From Whittaker, *How to Break Software*

Additional QuickTests from LAWST

Several of the tests we listed at LAWST (7th Los Altos Workshop on Software Testing, 1999) are equivalent to the attacks later published by Whittaker.

He develops the attacks well, and we recommend his descriptions.

LAWST generated several other quicktests, including some that aren't directly tied to a simple fault model.

Many of the ideas in these notes were reviewed and extended by the other LAWST 7 attendees: Brian Lawrence, III, Jack Falk, Drew Pritsker, Jim Bampos, Bob Johnson, Doug Hoffman, Chris Agruss, Dave Gelperin, Melora Svoboda, Jeff Payne, James Tierney, Hung Nguyen, Harry Robinson, Elisabeth Hendrickson, Noel Nyman, Bret Pettichord, & Rodney Wilson. We appreciate their contributions.

Additional QuickTests

Interference testing

- We look 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.

Additional QuickTests

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

Additional QuickTests

Interference: Change something 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

Additional QuickTests

Interference testing: Cancel

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

Additional QuickTests

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)
- For example,
  - 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.

Additional QuickTests

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.**

Additional QuickTests

Interference testing: Compete

Examples:

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

Additional QuickTests

Follow up recent changes

- Code changes cause side effects
  - Test the modified feature / change itself.
  - Test features that interact with this one.
  - Test data that are related to this feature or data set.
  - Test scenarios that use this feature in complex ways.

Additional QuickTests

Explore data relationships

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

Additional QuickTests

Explore data relationships

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

We discuss this table more in our lectures on combination testing.

Additional QuickTests

Explore data relationships (continued)

- Many possible relationships. For example,
  - V1 < V2+K (V1 is constrained by V2+K)
  - V1 = f(V2), where f is any function
  - V1 is an enumerated variable but the set of choices for V1 is determined by the value of V2
- Relations are often reciprocal, so if V2 constrains V1, then V1 might constrain V2 (try to change V2 after setting V1)
- Given the relationship,
  - Try to enter relationship-breaking values everywhere that you can enter V1 and V2.
  - Pay attention to unusual entry options, such as editing in a display field, import, revision using a different component or program
- Once you achieve a mismatch between V1 and V2, the program's data no longer obey rules the programmer expected would be obeyed, so anything that assumes the rules hold is vulnerable. Do follow-up testing to discover serious side effects of the mismatch

Even More QuickTests (from Bach's Rapid Testing Course)

Quick tours of the program

- **Variability Tour:** Tour a product looking for anything that is variable and vary it. Vary it as far as possible, in every dimension possible.

  Exploring variations is part of the basic structure of Bach's testing when he first encounters a product.

- **Complexity Tour:** Tour a product looking for the most complex features and data. Create complex files.
- **Sample Data Tour:** Employ any sample data you can, and all that you can. The more complex the better.

» From Bach's Rapid Software Testing course

Even More QuickTests (from Bach's Rapid Testing Course)

- **Continuous Use:** While testing, do not reset the system. Leave windows and files open. Let disk and memory usage mount. You're hoping the system ties itself in knots over time.
- **Adjustments:** Set some parameter to a certain value, then, at any later time, reset that value to something else without resetting or recreating the containing document or data structure.
- **Dog Piling:** Get more processes going at once; more states existing concurrently. Nested dialog boxes and non-modal dialogs provide opportunities to do this.
- **Undermining:** Start using a function when the system is in an appropriate state, then change the state part way through (for instance, delete a file while it is being edited, eject a disk, pull net cables or power cords) to an inappropriate state. This is similar to interruption, except you are expecting the function to interrupt itself by detecting that it no longer can proceed safely.

Even More QuickTests (from Bach's Rapid Testing Course)

- **Error Message Hangover:** Make error messages happen. Test hard after they are dismissed. Developers often handle errors poorly. Bach once broke into a public kiosk by right clicking rapidly after an error occurred. It turned out the security code left a 1/5 second window of opportunity for me to access a special menu and take over the system.
- **Click Frenzy:** Testing is more than "banging on the keyboard", but that phrase wasn't coined for nothing. Try banging on the keyboard. Try clicking everywhere. Bach broke into a touchscreen system once by poking every square centimeter of every screen until he found a secret button.
- **Multiple Instances:** Run a lot of instances of the application at the same time. Open the same files.
- **Feature Interactions:** Discover where individual functions interact or share data. Look for any interdependencies. Tour them. Stress them. Bach once crashed an app by loading up all the fields in a form to their maximums and then traversing to the report generator.

Even More QuickTests (from Bach's Rapid Testing Course)

- **Cheap Tools!** Learn how to use InCtrl5, Filemon, Regmon, AppVerifier, Perfmon, Task Manager (all of which are free). Have these tools on a thumb drive and carry it around. Also, carry a digital camera. Bach carries a tiny 3 megapixel camera and a tiny video camera in his coat pockets. He uses them to record screen shots and product behaviors.
  - Elisabeth Hendrickson suggests several additional tools at http://www.bughunting.com/bugtools.html
- **Resource Starvation:** Progressively lower memory and other resources until the product gracefully degrades or ungracefully collapses.
- **Play "Writer Sez":** Look in the online help or user manual and find instructions about how to perform some interesting activity. Do those actions. Then improvise from them. Often writers are hurried as they write down steps, or the software changes after they write the manual.

Even More QuickTests (from Bach's Rapid Testing Course)

- **Crazy Configs:** Modify O/S configuration in non-standard or non-default ways either before or after installing the product. Turn on "high contrast" accessibility mode, or change the localization defaults. Change the letter of the system hard drive.
- **Grokking:** Find some aspect of the product that produces huge amounts of data or does some operation very quickly. For instance, look a long log file or browse database records very quickly. Let the data go by too quickly to see in detail, but notice trends in length or look or shape of the patterns as you see them.

Parlour Tricks are not Risk-Free

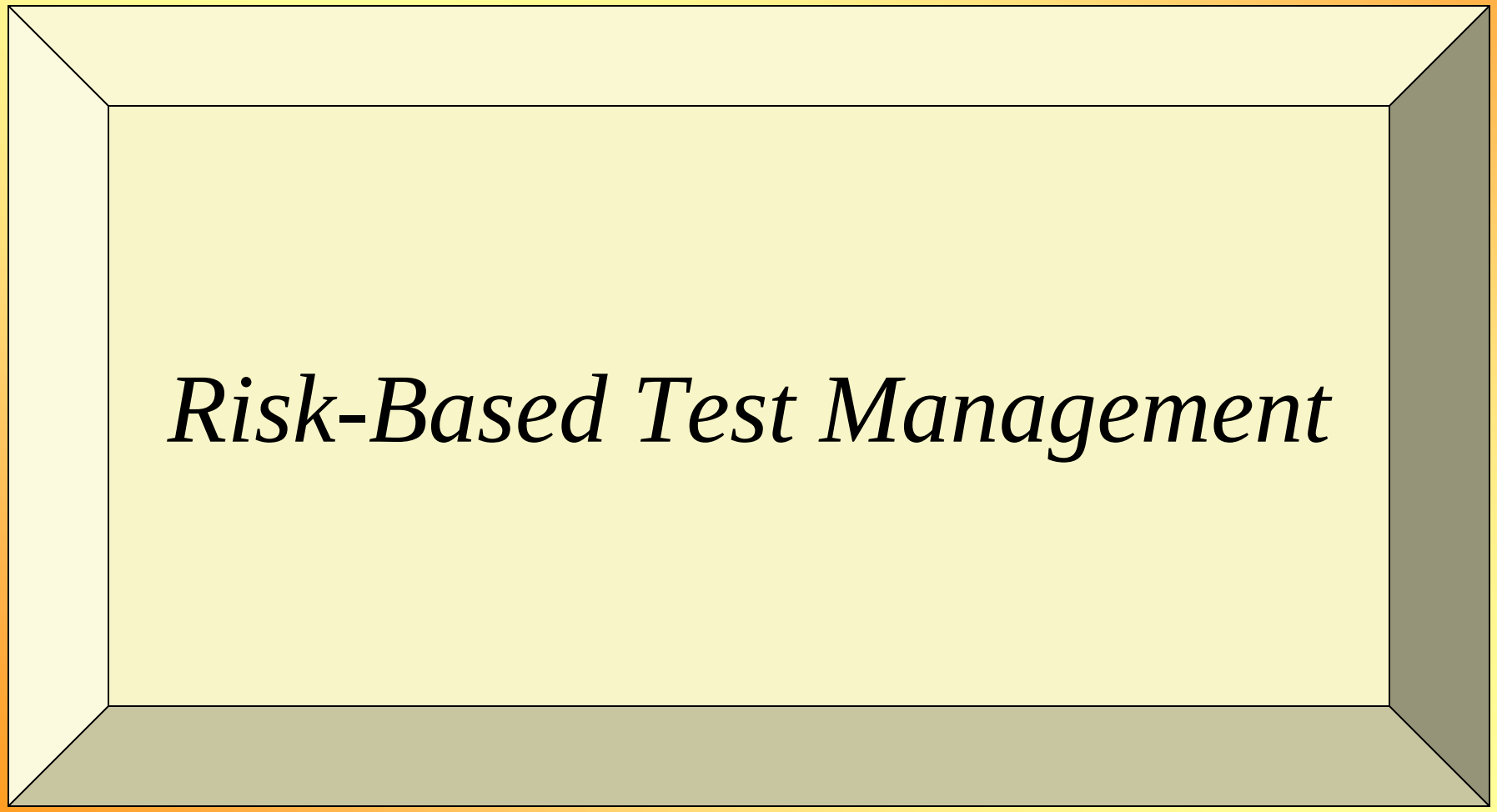
- These tricks can generate lots of flash in a hurry
  - The DOS disk I/O example
  - The Amiga clicky-click-click-click example
- As political weapons, they are double-edged
  - If people realize what you're doing, you lose credibility
  - Anyone you humiliate becomes a potential enemy
- Some people (incorrectly) characterize exploratory testing as if it were a collection of quicktests.
- As test design tools, they are like good candy
  - Yummy
  - Everyone likes them
  - Not very nutritious. (You never get to the deeper issues of the program.)


Operational Profiles

Operational profiles

- John Musa, *Software Reliability Engineering*
  - Based on enormous database of actual customer usage data
  - Accurate information on feature use and feature interaction
  - Given the data, drive reliability by testing in order of feature use
  - Problem: low probability, high severity bugs

- **Warning, Warning Will Robinson, Danger Danger!**
  - Unverified estimates of feature use / feature interaction
  - Drive testing in order of perceived use
  - Rationalize not-fixing in order of perceived use
  - Estimate reliability in terms of bugs not found



Introduction to risk-based testing

Risk: The possibility of suffering harm or loss

- In software testing, we think of risk on three dimensions:
  - *A way the program could fail*
  - *How likely it is that the program could fail in that way*
  - *What the consequences of that failure could be*

For testing purposes, the most important is:

- *A way the program could fail*

For project management purposes,

- *How likely*
- *What consequences*

Risk = Probability x Consequence?

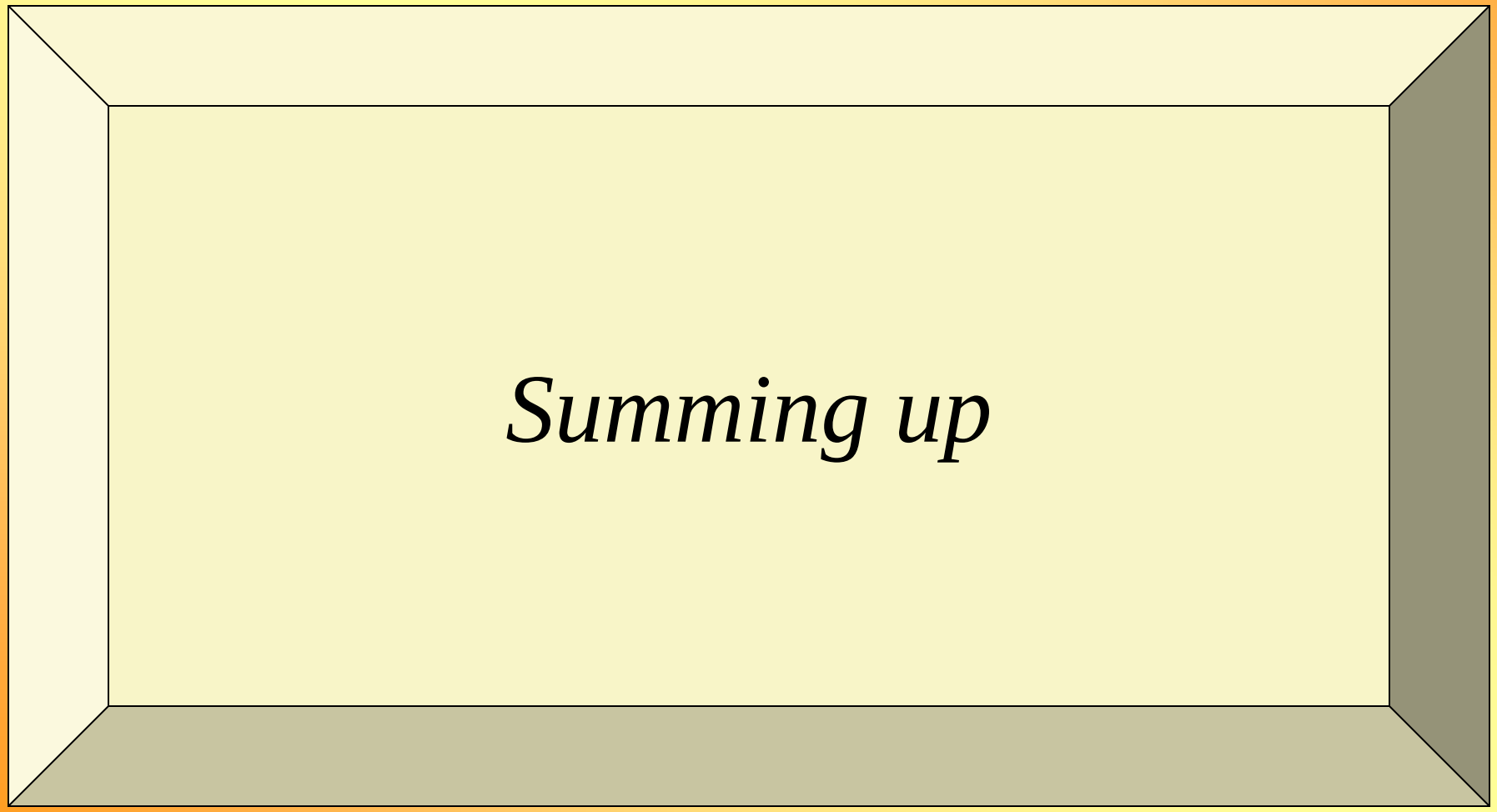
- List all areas of the program that could require testing
- On a scale of 1-5, assign a probability-of-failure estimate to each
- On a scale of 1-5, assign a severity-of-failure estimate to each
- Multiply estimated probability by estimated severity, this (allegedly) estimates risk
- Prioritize testing the areas in order of the estimated risk

Numerology?

- For a significant area of the product, what is the probability of error? (100%)
- What is the severity of this error? (Which error?)
- What is the difference between a level 3 and a level 4 and a level 5? (We don't and can't know, they are ordinally scaled.)
- What is the sum of two ordinal numbers? (undefined)
- What is the product of two ordinal numbers (undefined)
- What is the meaning of the risk of 12 versus 16? (none)

How "risk magnitude" can hide the right answer

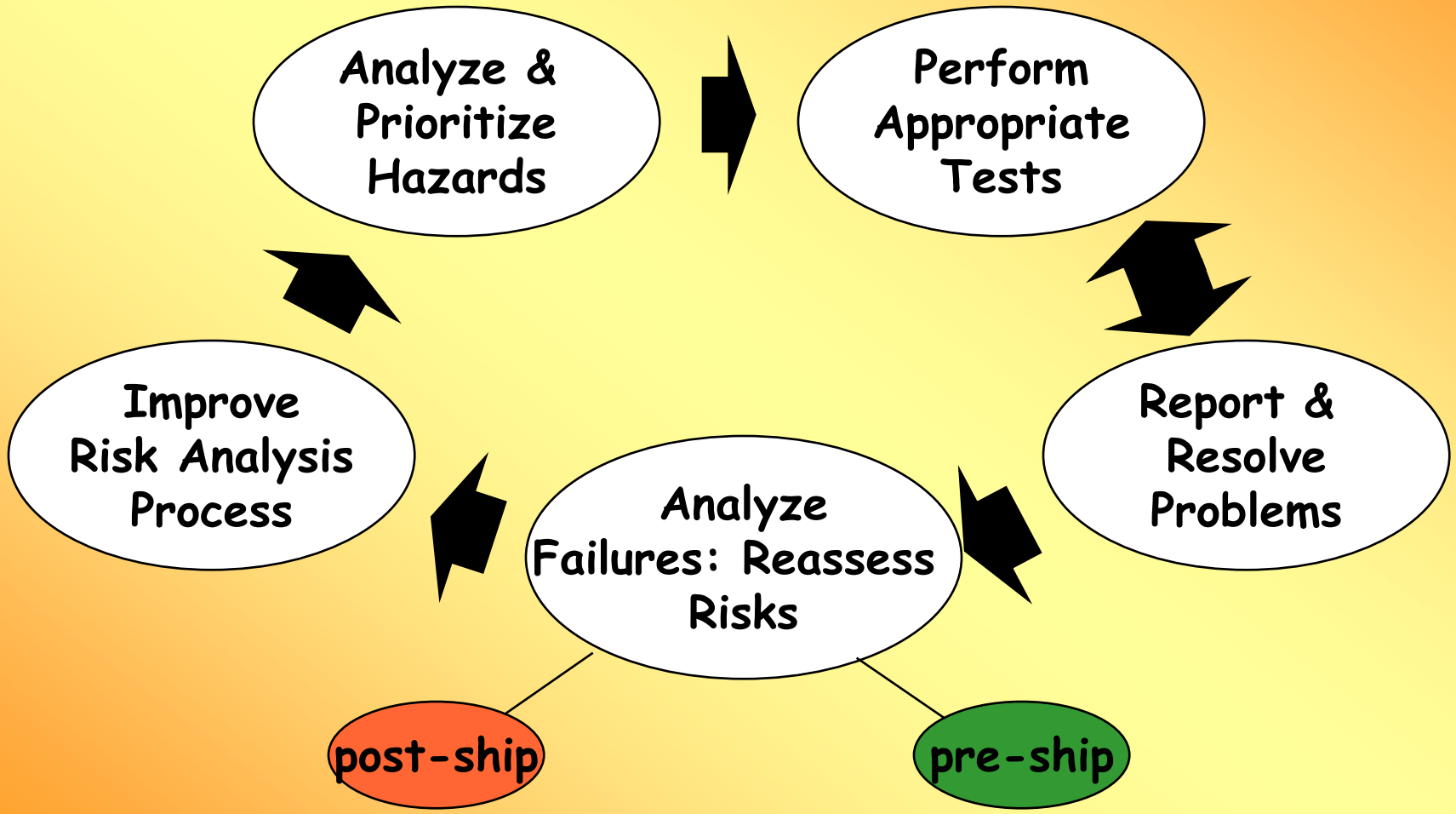
- A very severe event (impact = 10)…
- that happens rarely (likelihood = 1)…
- has less magnitude than 73% of the risk space.
- A 5 X 5 risk has a magnitude that is 150% greater.
- What about Borland's Turbo C++ project file corruption bug?
  - It cost hundreds of thousands of dollars and motivated a product recall. Yet it was a "rare" occurrence, by any stretch of the imagination. It would have scored a 10, even though it turned out to be the biggest problem in that release.


Summing up

Here's one way to do RBT

1. Get risk ideas:
   - From risk factors or from failure mode categories (use guideword heuristics)
2. For each important idea, *determine test activities, prepare tests (that have power against that idea)*, *shift resources* to gather information it. Otherwise, *escalate it*.
3. Maintain traceability between risks and tests.
4. Monitor and report the status of the risks as the project goes on and you learn more about them.
5. Assess coverage of the testing effort program, given a set of risk-based tests. Find holes in the testing effort.
6. If a risk is determined to be small enough, then stop testing against it.
7. On retesting an area, evaluate your tests experiences so far to determine what risks they were testing for and whether more powerful variants can be created.
8. Do at least *some* non-risk-based testing, to cover yourself in case your risk analysis is wrong.
9. Build lists of bug histories, configuration problems, tech support requests and obvious customer confusions -- deepen your lists of failure modes

Risk-driven testing cycle



Risk-based testing

- Testing software is like testing scientific theories:
  - Karl Popper, in his famous essay *Conjectures and Refutations*, lays out the proposition that a scientific theory gains credibility by being subjected to (and passing) harsh tests that are intended to refute the theory.
  - We can gain confidence in a program by testing it harshly (if it passes the tests).
  - Subjecting a program to easy tests doesn't tell us much about what will happen to the program in the field.
- ***In risk-based testing, we create harsh tests for vulnerable areas of the program.***

Risk-based testing & scientific thinking

- Ability to pose useful questions
- Ability to observe what's going on
- Ability to describe what you perceive
- Ability to think critically about what you know
- Abitlity to recognize and manage bias
- Ability to form and test conjectures
- Ability to keep thinking despite already knowing
- Ability to analyze someone else's thinking

Risk-based testing: Some papers of interest

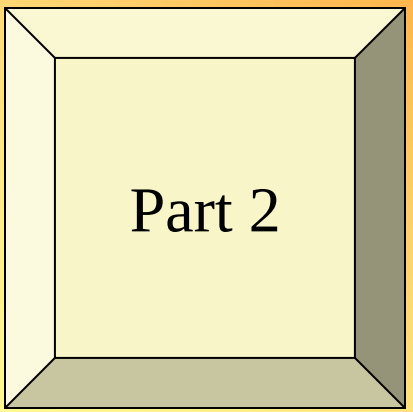
- Stale Amland, *Risk Based Testing*, http://www.amland.no/WordDocuments/EuroSTAR99Paper.doc
- James Bach, *Reframing Requirements Analysis*
- James Bach, Risk and Requirements- Based Testing
- James Bach, James Bach on Risk-Based Testing
- Stale Amland & Hans Schaefer, Risk based testing, a response (at http://www.satisfice.com)
- Stale Amland's course notes on Risk-Based Agile Testing (December 2002) at http://www.testingeducation.org/coursenotes/amland_stale/cm_200212_exploratorytesting
- Carl Popper, Conjectures & Refutations
- James Whittaker, *How to Break Software*
- Giri Vijayaraghavan's papers and thesis on bug taxonomies, at http://www.testingeducation.org/articles

Black Box Software Testing

Fall 2005

RISK-BASED TESTING

by
Cem Kaner, J.D., Ph.D.
Professor of Software Engineering
Florida Institute of Technology
and
James Bach
Principal, Satisfice Inc.



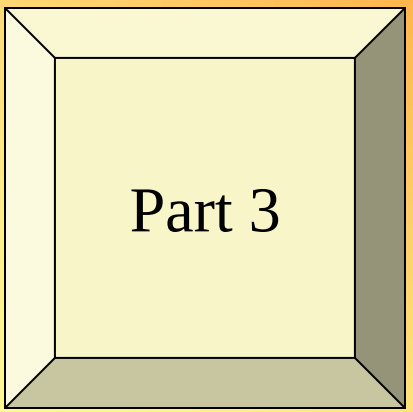
These notes are partially based on research that was supported by NSF Grant EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers." Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of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

Black Box Software Testing

Fall 2005

RISK-BASED 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.

Additional QuickTests Hey, look! I changed the slide!

Interference: Change something 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