Black Box Software Testing: Introduction to Test Design: A Survey of Test Techniques

Cem Kaner, J.D., Ph.D.
Professor of Software Engineering: Florida Tech

Rebecca L. Fiedler, M.B.A., Ph.D.
President: Kaner, Fiedler & Associates

These notes are partially based on research that was supported by NSF Grants EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers" and CCLI-0717613 "Adaptation & Implementation of an Activity-Based Online or Hybrid Course in Software Testing." 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.

Notice …

The practices recommended and discussed in this course are useful for an introduction to testing, but more experienced testers will adopt additional practices.

I am writing this course with the mass-market software development industry in mind. Mission-critical and life-critical software development efforts involve specific and rigorous procedures that are not described in this course.

Some of the BBST-series courses include some legal information, but you are not my legal client. I do not provide legal advice in the notes or in the course.

If you ask a BBST instructor a question about a specific situation, the instructor might use your question as a teaching tool, and answer it in a way that s/he believes would "normally" be true but such an answer may be inappropriate for your particular situation or incorrect in your jurisdiction. Neither I nor any instructor in the BBST series can accept any responsibility for actions that you might take in response to comments about technology or law made in this course. If you need legal advice, please consult your own attorney.

… And thanks

The BBST lectures evolved out of practitioner-focused courses co-authored by Kaner & Hung Quoc Nguyen and by Kaner & Doug Hoffman (now President of the Association for Software Testing), which then merged with James Bach's and Michael Bolton's Rapid Software Testing (RST) courses. The online adaptation of BBST was designed primarily by Rebecca L. Fiedler.

Starting in 2000, the course evolved from a practitioner-focused course through academic teaching and research largely funded by the National Science Foundation. The Association for Software Testing served (and serves) as our learning lab for practitioner courses. We evolved the 4-week structure with AST and have offered over 30 courses to AST students. We could not have created this series without AST's collaboration.

We also thank Scott Allman, Jon Bach, Scott Barber, Bernie Berger, Ajay Bhagwat, Rex Black, Michael Bolton, Fiona Charles, Jack Falk, Elizabeth Hendrickson, Kathy Iberle, Bob Johnson, Karen Johnson, Brian Lawrence, Brian Marick, John McConda, Greg McNelly, Melora Svoboda, dozens of participants in the Los Altos Workshops on Software Testing, the Software Test Mangers' Roundtable, the Workshops on Heuristic & Exploratory Techniques, the Workshops on Teaching Software Testing, the Austin Workshops on Test Automation and the Toronto Workshops on Software Testing and students in over 30 AST courses for critically reviewing materials from the perspective of experienced practitioners. We also thank the many students and co-instructors at Florida Tech, who helped us evolve the academic versions of this course, especially Pushpa Bhallamudi, Walter P. Bond, Tim Coulter, Sabrina Fay, Anand Gopalakrishnan, Ajay Jha, Alan Jorgensen, Kishore Kattamuri, Pat McGee, Sowmya Padmanabhan, Andy Tinkham, and Giri Vijayaraghavan.

GETTING THESE SLIDES

Download them from …

www.testingeducation.org/BBST

in the section on Test Design

Instructor training available at …

http://www.testingeducation.org/BBST

BBST LEARNING OBJECTIVES

- Understand key testing challenges that demand thoughtful tradeoffs by test designers and managers.
- Develop skills with several test techniques.
- Choose effective techniques for a given objective under your constraints.
- Improve the critical thinking and rapid learning skills that underlie good testing.
- Communicate your findings effectively.
- Work effectively online with remote collaborators.
- Plan investments (in documentation, tools, and process improvement) to meet your actual needs.
- Create work products that you can use in job interviews to demonstrate testing skill.

COURSE OBJECTIVES

This is an introductory survey of test design. The course introduces students to:

- many techniques at a superficial level (what the technique is),
- a few techniques at a practical level (how to do it),
- ways to mentally organize this collection,
- using the Heuristic Test Strategy Model for test planning and design, and
- using concept mapping tools for test planning.

We don't have time to develop your skills in these techniques. Our next courses will focus on one technique each. THESE will build deeper knowledge and skill, technique by technique.

Any of these techniques can be used in a scripted way or an exploratory way.

CHANGING EMPHASES ACROSS THE COURSES

	Foundations	Bug advocacy	Test design	Domain testing	Spec-based testing	Scenario-based testing
Greatest emphasis	Course skills	Testing skills	Testing knowledge	Testing skills	Testing skills	Testing skills
↕	Testing knowledge	Testing knowledge	Learning skills	Testing knowledge	Learning skills	Learning skills
	Social skills	Social skills	Testing skills	Learning skills	Testing knowledge	Social skills
	Computing fundamentals	Learning skills	Course skills	Computing fundamentals	Social skills	Testing Knowledge
	Learning skills	Course skills	Social skills	Social skills	Course skills	Computing fundamentals
Least emphasis	Testing skills	Computing fundamentals	Computing fundamentals	Course skills	Computing fundamentals	Course skills

Changing emphases

- **Course Skills:** Working effectively in online courses. Taking tests. Managing your time.
- **Social Skills:** Working together in groups. Peer reviews. Using collaboration tools (e.g. wikis).
- **Learning Skills:** Using lectures, slides, and readings effectively. Searching for supplementary information. Using these materials to form and defend your own opinion.
- **Testing Knowledge:** Definitions. Facts and concepts of testing. Structures for organizing testing knowledge.
- **Testing Skills:** How to actually do things. Getting better (through practice and feedback) at actually doing them.
- **Computing Fundamentals:** Facts and concepts of computer science.

OVERVIEW OF THE COURSE

	Technique	Context / Evaluation
1	Function testing & tours.	A taxonomy of test techniques.
2	Risk-based testing, failure mode analysis and quicktests	Testing strategy. Introducing the Heuristic Test Strategy Model.
3	Specification-based testing.	(… work on your assignment …)
4	Use cases and scenarios.	Comparatively evaluating techniques.
5	Domain testing: traditional and risk-based	When you enter data, any part of the program that uses that data is a risk. Are you designing for that?
6	Testing combinations of independent and interacting variables.	Combinatorial, scenario-based, risk-based and logical-implication analyses of multiple variables.

LECTURE 1 READINGS

Required reading

- Kelly, M.D. (2005). Taking a tour through test country. http://michaeldkelly.com/pdfs/Taking a Tour Through Test Country.pdf
- Kaner, C., Bach, J., & Pettichord, B. (2001). Lessons Learned in Software Testing: Chapter 3: Test Techniques. http://media.techtarget.com/searchSoftwareQuality/downloads/Lessons_Learned_in_SW_testingCh3.pdf

Recommended reading

- Bolton, M. (2009). Of testing tours and dashboards. http://www.developsense.com/blog/2009/04/of-testing-tours-and-dashboards
- Bolton, M. (2006). The factors of function testing. http://www.developsense.com/articles/2006-07-TheFactorsOfFunctionTesting.pdf
- Copeland, L. (2004). A Practitioner's Guide to Software Test Design. Artech House
- Kelly, M.D. (2005). Touring Heuristic. http://www.michaeldkelly.com/archives/50

Your class might use other readings, but we had these in mind when creating this lecture.

Function Testing & Tours

FUNCTION TESTING

- A **function** is something the product can do.
  - Functions might also be called
    - Features
    - Commands
    - Capabilities
- In **function testing**, testers
  - Focus testing on each function (or subfunction), one by one.

Identifying functions

Testers don't come to a program knowing everything about it. They have to learn what they're testing.

To discover a product's functions:

- Check specifications or the draft user manual
- Walk the user interface
- Try commands at the command line
- Search the program or resource files for command names

Walking the User Interface

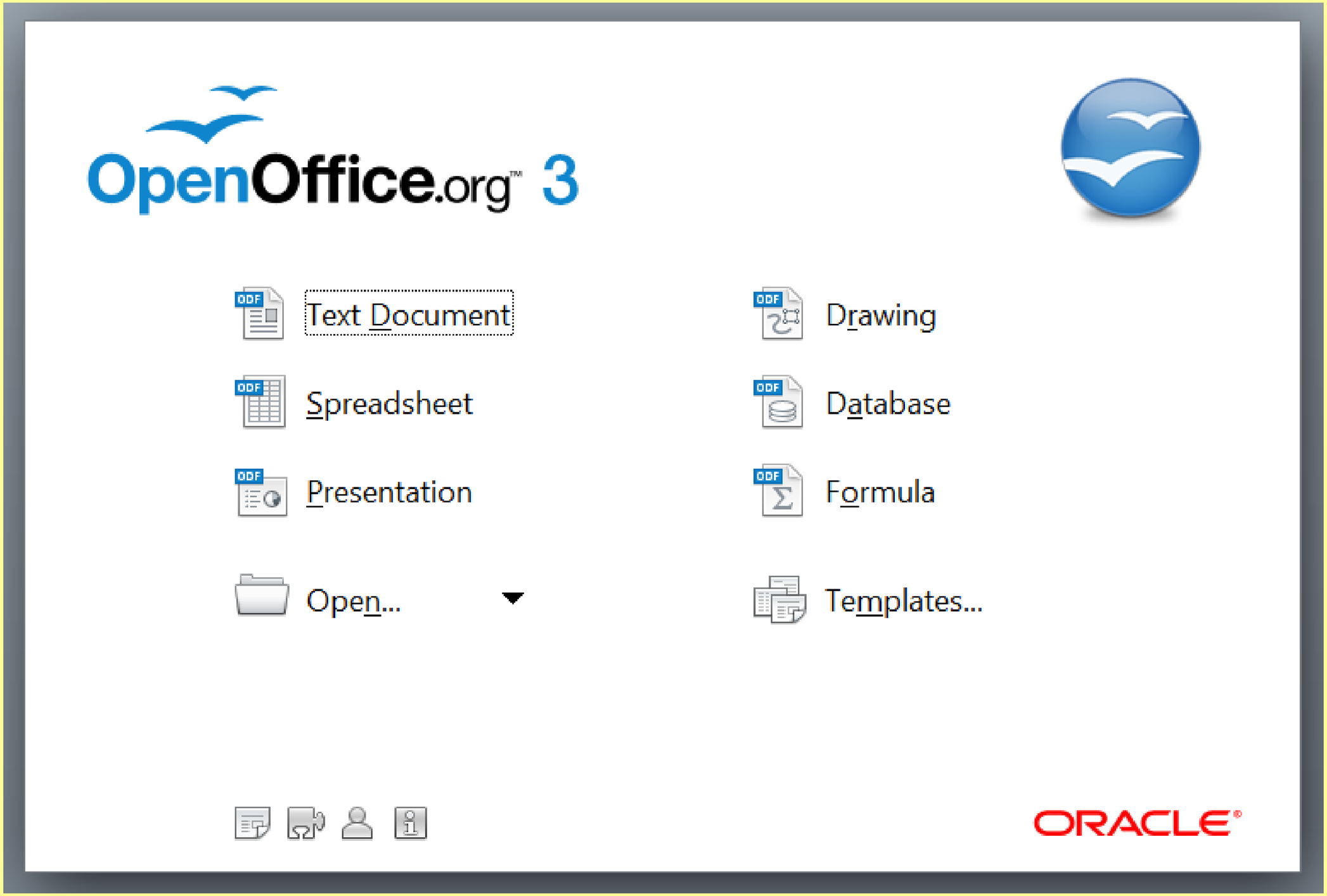
Here's a demonstration of walking the user interface, with OpenOffice Writer. The goal is to find every feature that you can reach through the user interface. To do that, we'll:

- Pull down menus and bring up dialogs
- Look for state-dependent dialogs or features
- Right-click everywhere to bring up context-sensitive menus
- Look for option settings that reveal new features
- And so on.

This is also called a feature tour.

A feature tour might also include looking at documentation, reverse engineering the product and any other activities that could help you quickly catalog the program's features.

THE DEMONSTRATION…



Tours and exploration

- That was an example of a tour
- James Bach, Elisabeth Hendrickson and Michael Kelly started describing the use of a variety of tours in exploratory testing in the 1990's (Kelly, 2005; Bolton, 2009).
- A tour is an exploration of a product that is organized around a theme. The theme I just illustrated focused on features.
- Think of touring as functionally similar to a structured brainstorming approach--excellent for surfacing a collection of ideas, that you can then explore in depth, one at a time.

Tours and Exploration

- The core of exploration is learning. Touring is one of several starting points for exploratory testing:
  - You might discover bugs (or other quality-relevant information) during a tour;
  - You might discover bugs when you follow up a tour with a deeper set of tests guided by your tour results.

Explaining why he finds tours valuable, Michael Bolton writes, "One of the challenges I've encountered in early exploration is managing focus--logging or noting bugs without investigating them; recognizing risks or vulnerabilities and not stopping, but noting them instead. Screen recording tools can backstop notes, too."

A TOUR YIELDS AN INVENTORY

- The result of a tour is an inventory: a list of things of the type that we're interested in.
- A "complete" tour yields an exhaustive list. (Most tours are incomplete but useful.)
- Later, you can test everything on the list, to some intentional degree of thoroughness.
- In our example, we created an inventory of the program's functions. Testing from that inventory is called **function testing**.

Touring lays the groundwork for coverage-oriented testing

- Function testing illustrates a coverage-driven test technique. Start with an inventory of functions, and organize your testing to achieve any level of *function coverage* that you choose.
- Given an inventory of error messages, you could organize your testing around *error-message coverage*.
- Given an inventory of variables, you could organize your testing around *variable coverage*.

Coverage measures how much of a certain type of testing we've completed, compared to the total number of possible tests of this type.

SUGGESTIONS FOR TOURING

- Testers can tour together:
  - Touring in pairs is often more productive than touring for twice as long, alone.
  - New testers will benefit from a tour guide.
- Testers can split tours across many sessions:
  - There is no need to complete a tour in one day or one week.
  - It's OK to tour for an hour or two, do some other tasks, then pick up the tour a few days later.

There are many types of tours

James Bach, Elisabeth Hendrickson and Michael Kelly have described several types of tours (Kelly, 2005; Bolton, 2009). Here is a collection of examples, based on their work:

- **Feature tour**:
  - This is what I've been calling the function tour
  - The tour goal: Find out what the program can do. Find all the features, controls and command line options.

There are many types of tours

Feature tour: Some people do sub-tours:

- **Menus and Windows Tour**: Find all the menus (main and context menus), menu items, windows, toolbars, icons, and other controls.

For touring menus and windows in Windows, Michael Bolton recommends Resource Hunter at

http://www.boilsoft.com/rchunter.html

There are many types of tours

Feature tour: Some people do sub-tours:

- **Mouse and Keyboard Tour**: Find all the things you can do with a mouse and keyboard. Click on everything. Try all the keys on the keyboard, including F-keys, Enter, Tab, Escape, Backspace and combinations with Shift, Ctrl, and Alt.

There are many types of tours

Transactions tour: A transaction is a complete task.

- For example, go to the store, buy something (including paying for it) is a complete task
- A transaction typically involves a sequence of many functions
- There is no bright line between "transactions" and "features", but the mindset of the tester doing the tour is a bit different
  - *What can I do with this program?* (transaction)
  - *What are the program's commands?* (feature / function)

There are many types of tours

Error Message Tour:

- List every error message. Ask programmers for a list, and look for a text file that stores program strings (including error messages)
  - Process Explorer, a System Internals tool available on Microsoft's Web site, includes functions for dumping a program's strings
  - Also handy: Resource Hunter, www.boilsoft.com.
- List every condition that you think *should* throw an error message
  - Highlight cases that should yield an error message but do not

There are many types of tours

Variables tour: What can you can change in the application?

- Anything you can change is a variable. Find them all.
- What values can each variable take?
- What are the variables' subranges and subrange-boundaries?
- Some variables (such as many variables that are set with check boxes) enable, disable or constrain the values of other variables. What changes in one variable will cause changes in other variables?

There are many types of tours

Data tour: What are the data elements of the application?

- Some of these are variables
- Others might be constants, or information the program reads from disk or obtains from other applications.

The variables tour and the data tour are obviously related, but their emphasis is different. The variables tour identifies the variables in the program. The data tour considers what values are fed to those variables and where those values come from.

There are many types of tours

Sample Data Tour: What samples are available?

- Consider data from using or testing a previous version of this application, or an application that should interoperate with this one.
- One caution: How will you determine whether the program handles these data correctly?

There are many types of tours

File Tour: What files are used by this program and where are they?

- What's in the folder where the program's .EXE file is found? What other folders contain application data? Check out your system's directory structure, including folders that might be available only to the system administrator.
- Look for READMEs, help files, log files, installation scripts, .cfg, .ini, .rc files. Look at the names of .DLLs, and extrapolate on the functions that they might contain or the ways in which their absence might undermine the application.

There are many types of tours

Structure tour: What is included in the complete product

- Code
- Data
- Interfaces
- Documentation
- Hardware
  - Security devices required for access
  - Cables
- Packaging
- Associated web sites or online services
- Anything you can test…

There are many types of tours

Operational modes tour: "An operational mode is a formal characterization of the status of one or more internal data objects that affect system behavior." (Whittaker, 1997, 120; Jorgensen, 1999; El-Far, 1999).

- In practice, in black-box state-model based testing, an operational mode is a visible state of the program.
- Identifying all the operational modes (states) is one of the core tasks of state-model based testing, followed by identifying the transitions (from State X, you can go directly to States Y and Z) and then testing all the transitions.

For excellent introductions to state-model based testing, see Harry Robinson's papers (see the bibliography at the end of the slide set).

There are many types of tours

Sequence tour: A sequence (or a sub-path) is a set of actions that take you from one state to another. For example,

Open a document

→ Print the document

→ Change the document

→ Start to save the document

This is one sequence that takes you to the Save-Document state. Several others could take you to the same state.

- What are they?
- Which ones are interesting?

As we use the term here, a sequence is a cognitively coherent path through the program (someone would do this on purpose) that might involve many state transitions.

There are many types of tours

Claims tour: What claims does the vendor make about the product?

- Find published claims in manuals, help, tutorials, and advertisements.
- Find unpublished claims in internal specifications and memos that make promises or define the product's intent.

There are many types of tours

Benefits tour: What are the benefits of this application?

- Your task is not to find bugs; it is to discover what the program will provide (what's valuable about this program) once it is fully working.
- This is a useful starting point for testers. You can communicate much more effectively about a product if you understand what it should be good for.

There are many types of tours

Market Context tour: What is the market context for this product?

- Who are the competitors?
- Among the competitors, are there cheaper ones? Less capable ones? More capable? More expensive?
- Why would someone buy this one instead of the other ones? Why would they buy the other ones?

There are many types of tours

User tour: What would users do with this program?

- Imagine five different types of users of this application.
- What would they do with it?
- What information would they want from it?
- How would they expect its features to work?

There are many types of tours

Life history tour: Consider an object in the system. (For example, consider a checking account in a banking application.)

- When and how can the program create it?
- How can the program change it?
- How is it used?
- What does it interact with?
- When does the program deactivate, discard or destroy it?
- After the object is terminated, is any record kept of it, for future reference?

You can create many scenarios to reflect different potential life histories for the same types of objects. See Kaner (2003), *An introduction to scenario testing* for discussion of life histories and 15 other themes for creating scenarios, each of which could yield its own type of tour.
http://www.kaner.com/pdfs/ScenarioIntroVer4.pdf

There are many types of tours

Configuration tour:

- What program settings can you change?
- Which will persist if you exit the program and reopen it?
- What operating system settings will affect application performance?

There are many types of tours

Interoperability tour: What does this application interact with?

- Find every aspect of this application that communicates with other software (including device drivers, competing applications, and external clients or servers).
- "Communication" includes any aspect of the application that creates data that other software will use or reads data saved by other software.

There are many types of tours

Compatibility tour:

- What devices should your program work with?
- What platforms (operating system and other system software) should your program run on?

There are many types of tours

Testability tour: What support for testing is built into this application?

- Find all the features you can use as testability features.
- Identify available tools that can help in your testing.

The common result of a testability tour is a request for more testability features.

Example: the software under test trades messages with another program to get its data. In the testability tour, you might realize it would be useful to see those messages. So you ask for a new feature, a log file that saves the contents of every message between these programs.

There are many types of tours

Specified-risk tour:

- Build a catalog of risks.
- Imagine a way that the program could fail and then search for all parts of the program that **might** fail this way.
  - The extreme-value tour is an example of this type of tour.
- Or walk through the program asking, at each point, "what could go wrong here"?
  - Then sort the failure ideas into categories and for each category, try to imagine other parts of the program that could fail in the same way.

In my experience, creating risk catalogs involves many iterations of touring, categorizing, and brainstorming. See Kaner et al (2000b); Vijayaraghavan & Kaner (2002, 2003); Jha (2007).

There are many types of tours

Extreme Value tour: What might cause problems for your variables? *What are the programmers' assumptions about incoming data?*

- Zero is usually interesting.
- Try small numbers where large ones are expected; negatives where positive ones are expected; huge ones where modestly-sized ones are expected; non-numbers where numbers are expected; and empty values where data is expected.
- Try potentially unexpected units, formats, or data types.

Some tours seem more like taking a simple inventory; others use more aggressive testing.
What's common to the tours is that you do what's necessary to identify a set of information of a desired type. Test to the level of creativity and depth needed to uncover that information.
You can structure deeper testing, guided by the tour's results, later.

There are many types of tours

Complexity tour: What are the most complex aspects of the application?

- Along with features, consider challenging data sets.
- Find the five (N) most complex tasks or aspects of the application.

Individual differences are to be expected

Consider the **complexity tour**

- The underlying hypothesis is that complex aspects of the program are more vulnerable to failure, malware attack, or user dissatisfaction.
- The touring tester looks for cases in which the program requires a long sequence of user actions or combines many functions to achieve one result, or uses many variables at once.

This tour might be rare in practice and the follow-up (risk-based testing) will be hard work. But someone who has the knowledge and skill to do this will learn things about the software's potential weaknesses (and ways to navigate the program) that less challenging tours will miss.

Diversity and exploration

Exploratory Testing is about using empirical methods (tests) to learn new things about the software.

Testers run new tests all the time – testers run new ***types*** of tests all the time – to gain new knowledge.

Each type of tour has testers explore the program from a different perspective, looking for different types of information.

No one will use every type of tour. But people have different interests, backgrounds and skills. A greater diversity of <u>available</u> tours enables greater diversity in testing. Some testers will explore a program in very different ways from other testers.

Back to Function Testing

Function testing: Key objective

- In **function testing**, testers
  - Focus on individual functions, testing them one by one.
- Goal (not necessarily achieved):
  - Test each function of the product

Creating a function list

To do function testing well, testers often create a **function list.**

- A function list is an outline of the program's capabilities
- You can create the function list while doing the feature tour.

Creating a function list

The best tools for function lists are concept-mapping programs:

- http://www.graphic.org/
- http://www.mindtools.com/mindmaps.html
- http://users.edte.utwente.nl/lanzing/cm_bibli.htm
- http://www.innovationtools.com/resources/mindmapping.asp

Tool list:

- http://en.wikipedia.org/wiki/List_of_concept_mapping_and_mind_mapping_software

I prefer:

- Mind Manager: http://www.mindjet.com
- NovaMind: http://www.novamind.com
- Inspiration, http://www.inspiration.com
- XMind, http://www.xmind.net

We'll create a function list with the XMind concept-mapping tool in our specification-based testing assignment.

Using function testing in early testing of the product

- Test sympathetically
  - What are the developers trying to achieve with this program?
  - What would it do if it worked properly?
  - Why would people want to use this?
  - What are its strengths?
- You can use the process of creating the function list to gain a sympathetic overview of the program's capabilities

"This sympathetic approach is really important even though some testers will find its opposite (charge in and find bugs) hard to break.
"Jon Bach pointed out to me that in early exploration, he tests to find benefits. I thought this was weird, until he pointed out that finding and recording bugs caused him to lose focus on touring the product and building his model of it. I've found that getting over the habit of driving straight to the bugs is tough. It's a dramatically different rhythm that for me required practice."

-- Michael Bolton

Using function testing in early testing of the product

- If you run complex tests early, one broken function can block you from running an entire test
  - Blocking bugs might prevent you from discovering a broken feature until late in the project
- Fast scan for serious problems
  - Some aspects of the product are so poorly designed or so poorly implemented that they must be redone
    - Better to realize this early
    - Pointless to test an area in detail if it will be replaced

Using function tests for smoke testing

Smoke testing (aka build-verification testing):

- Most tests in a smoke-test suite are function tests
- Relatively small set of tests run whenever there is a new build.
- Question is whether the build is worth more thorough testing.

In some companies, testers design the smoke tests and give them to the programmers, who run the smoke tests as part of their build process.

History of "smoke test":

- **New circuit board (or other electronic equipment)**
- **Apply power**
- **If you get smoke: stop testing**

Using function testing beyond early testing

- You probably run more complex tests that involve several features.
- The function list can serve as a coverage guide for this testing. Are you reaching:
  - every feature?
  - every subfunction of every feature?
  - every option of every feature?

Using function testing as your main technique

IF you use function testing as your main test technique:

- You will test the functions one by one (tests are focused on one function at a time), but you test each one as thoroughly as you can.
- Extend your function list to include more detail (see "**The fully-detailed function list**", next slide).
- Focus your testing on the detailed list
  - Run the specific tests suggested by the list
  - Explore freely, using the list as a foundation, not as a limit on scope.

We do not recommend using function testing as your main technique.

However, some companies do this. This slide is about what they do.

THE FULLY-DETAILED FUNCTION LIST

- Category of functions (e.g. Editing)
  - Individual functions (e.g. Cut, Paste, Delete)
    - Inputs to the function
      - » Variable
        - » Maximum value
        - » Minimum value
        - » Other special cases
    - Outputs of the function
    - Possible scope of the function (e.g. Delete word, Delete paragraph)
    - Options of the function (e.g. configure the program to Delete the contents of a row of a table, leaving a blank row versus Delete the row along with its contents)
    - Error cases that might be reached while using this function
    - Circumstances under which the function behaves differently (e.g. deleting from a word processor configured to track and display changes or not to track changes)

The fully-detailed function list

Add notes to describe

- how you would know
  - if the function works
  - if the function does not work.
- how the function is used
  - What is the result (the consequence) of running this function?
- environmental variables that may constrain the function under test.

No one (that we know of) creates function lists with all this detail for every function. But many people find it useful to add notes like these for a few functions.

Risks of using function testing as your main technique

Function testing is primarily a test of capability of individual units of the software.

It de-emphasizes:

- Interactions of features
- Special values of data, and interactions of values of several variables
- Missing features
- User tasks—whether the customer can actually achieve benefits promised by the program
- Interaction with background tasks, effects of interrupts
- Responsiveness and how well the program functions under load
- Usability, scalability, interoperability, testability, etc.

You can try to stretch function testing's scope—for example with the much more heavily detailed function lists—but other techniques are naturally focused on concerns that you must stretch function testing to (try to) reach.

An overview of test techniques

Test techniques: Defined

A technique is:

- "A body of technical methods (as in a craft, or in scientific research)"
- "A method of accomplishing a desired aim."

Merriam-Webster dictionary

- “the body of specialized procedures and methods used in any specific field, esp. in an area of applied science.
- method of performance; way of accomplishing.”

Reference.com

Test techniques: Defined

- Techniques involve **skill**: You get better at applying a technique as you gain experience with it.
- Techniques are more **action** than theory: You might need some theoretical background to understand a technique, and a technique might apply theoretical knowledge, but the technique itself is about **how to do** a type of testing.
- Techniques are **different from each other**: Any one technique will be more effective obtaining some types of knowledge (e.g. some types of bugs) but less effective for others.

A test technique is a method of designing, running and interpreting the results of tests.

Approaches vs. techniques

Some testers call **exploratory testing** and **scripted testing** test techniques.

You can use any technique in

- an exploratory way or
- a scripted way or
- a way that includes both exploratory and scripted elements.

Exploration and script-following reflect broad visions about the best way to organize and do testing, not specific tactics for designing individual tests.

Therefore we call them approaches rather than techniques. (Some people would call them *competing paradigms).*

Driving ideas behind many techniques

The "pure" vision of a technique is a way of doing something.

In practice, most test techniques specify only some of the how-to-do-it details, leaving the others open.

The guidance given by a testing technique might focus on any of the following:

- Scope
- Coverage
- Testers
- Risks (potential problems)
- Activities
- Evaluation / oracles
- Desired result

Every test addresses all of these. A specific technique typically addresses 1 to 3 of them, leaving the rest to be designed into the individual test.

Driving ideas behind many techniques

- **Scope:** what gets tested. Example: in function testing, test individual functions.
- **Coverage**: intended extent of testing. Example: in *function testing*, you test every function. We'll typically analyze scope (what to test) and coverage (how much of it) together.
- **Testers**: who does the testing. Example: *user testing* is focused on testing by people who would normally use the product.
- **Risks**: potential problem you're testing for. Example: *boundary errors*.
- **Activities**: how you actually *do* the tests. Example: *all-pairs testing* specifies how you combine conditions to obtain test cases.
- **Evaluation / Oracle**: how to tell whether the test passed or failed. Example: *function equivalence testing* relies on comparison to a reference function.
- **Desired result**: testing with a tightly-defined objective. Example: *build verification testing* checks whether the build is stable enough for more thorough testing.

Function testing as a technique

Guidance from function testing:

- **Scope**: Focus on individual functions, testing them one by one.
- **Coverage**: Test every function (or a subset that is a knowable proportion)

What function testing doesn't specify:

- **Testers**: Who does the testing
- **Risks**: What bugs we're looking for
- **Activities**: How to run the tests
- **Evaluation / oracles**: How to evaluate the test results

This discussion of our approach to techniques is based on Kaner, Bach & Pettichord (2001) *Lessons Learned in Software Testing*.

Classifying the techniques

Many techniques implement more than one underlying idea. Because of this, how you classify a technique depends on what you have in mind when you use it.

For example, feature integration testing:

- is **coverage-oriented** if you are checking whether every function behaves well when used with other functions.
- is **risk-oriented** if you have a theory of error for interactions between functions.

Examples

Here's a loosely organized collection of some (far from all) test techniques.

You cannot possibly absorb all this information in this course.

- While you take the course, this set will give you a sense of the variety of techniques;
- Later, when you are testing programs, we hope you try techniques from the list. We've provided references for the techniques to facilitate this.

Remember: exam questions will be drawn from the study guide. Before you panic at all the detail in these slides, look over the questions. We don't expect you to memorize a lot of detail.

Coverage-based techniques focus on what gets tested

- Function testing
- Feature or function integration testing
- Tours
- Equivalence class analysis
- Boundary testing
- Best representative testing
- Domain testing
- Test idea catalogs
- Logical expressions
- Multivariable testing
- State transitions
- User interface testing
- Specification-based testing
- Requirements-based testing
- Compliance-driven testing
- Configuration testing
- Localization testing

In principle, a coverage-based technique sets you up to run <u>every</u> test of a given type.
In practice, you probably won't run <u>every</u> test of <u>any</u> type, but you might measure your coverage of that type of testing.

Tester-based techniques focus on who does the testing

- User testing
- Alpha testing
- Beta testing
- Bug bashes
- Subject-matter expert testing
- Paired testing
- Eat your own dogfood
- Localization testing

There's a mystique in designing a technique around the type of person who tests. However, what they will actually do may have little to do with what you imagine will happen.

Risk-based techniques focus on potential problems

- Boundary testing
- Quicktests
- Constraints
- Logical expressions
- Stress testing
- Load testing
- Performance testing
- History-based testing
- Risk-based multivariable testing
- Usability testing
- Configuration / compatibility testing
- Interoperability testing
- Long sequence regression

Risk-based testing starts from an idea of how the program could fail. Then design tests that try to expose problems of that type.

Activity-based techniques focus on how you do the testing

- Guerilla testing
- All-pairs testing
- Random testing
- Use cases
- Scenario testing
- Installation testing
- Regression testing
- Long sequence testing
- Dumb monkey testing
- Load testing
- Performance testing

Because these focus on "how-to", these might be the techniques that most closely match the classical notion of a "technique."

Evaluation-based techniques focus on your oracle

- Function equivalence testing
- Mathematical oracle
- Constraint checks
- Self-verifying data
- Comparison with saved results
- Comparison with specifications or other authoritative documents
- Diagnostics-based testing
- Verifiable state models

Any time you have a well-specified oracle, you can build a set of tests around that oracle.
See our presentation of Hoffman's collection of oracles in BBST-Foundations.

Desired-result techniques focus on a specific decision or document

- Build verification
- Confirmation testing
- User acceptance testing
- Certification testing

You are doing document-focused testing if you run a set of tests primarily to collect data needed to fill out a form or create a clearly-structured report.

There are also glass-box techniques, such as…

- Unit tests
- Functional tests below the UI level
- Boundary testing
- State transitions
- Risk-based
- Dataflows
- Program slicing
- Protocol testing
- Diagnostics-driven testing
- Performance testing
- Compliance-focused testing
- Glass-box regression testing
- Glass-box decision coverage
- Glass-box path coverage

Most techniques that can be done black-box can also be used in glass-box testing.

What's different about glass box tests?

- Programmers can see the implementation tradeoffs, risks, and special cases in their code and write tests to focus on them.
- Programmers can capture state information that is invisible to black box testers.
- Programmers who create their own test libraries often write more testable code.
- Test execution is typically automated.
- Programmers typically run them many times—many tests are run every time the programmer compiles the software.
- The programmer sees failures immediately. There is no bug-report-writing delay or cost.
- Maintenance of these tests is probably cheaper and easier.

We won't further study these techniques in this course. We're simply reminding you that the black box techniques are just a subset.

COVERAGE-BASED TESTS

Coverage-based techniques focus on what gets tested

- Function testing
- Feature or function integration testing
- Tours
- Equivalence class analysis
- Boundary testing
- Best representative testing
- Domain testing
- Test idea catalogs
- Logical expressions
- Multivariable testing
- State-model-based testing
- User interface testing
- Specification-based testing
- Requirements-based testing
- Compliance-driven testing
- Configuration testing
- Localization testing

In principle, a coverage-based technique sets you up to run every test of a given type.
In practice, you probably won't run every test of any type, but you might measure your coverage of that type of testing.

Function testing

- Test each feature or function on its own.
- Scan through the product:
  - cover every feature or function
  - with at least enough testing to determine
    - what it does and
    - whether it is (basically) working.

Function testing gives you coverage of the features of the product.

Feature integration testing

- Test several features or functions together
- Typically, do this testing with functions that
  - will often be used together
    - Example: in a spreadsheet, sum part of a column, then sort data in the column. Sorting should change the sum if and only if you sort different values into the part being summed.
  - work together to create a result
    - Example: select a book, add it to a shopping cart, pay for the book

Feature integration testing gives you coverage of the interactions of the product's features.

Tours

A tour is a search through the product to create a collection of related information about the program.

For example, you can do:

- A feature tour, to find every feature
- A variable tour, to find every user-changeable variable
- An output tour, to find every variable, every report, and every user-visible message the program can create

Tours provide a basis for coverage-based testing. Create a list of things to test with a tour (e.g. a list of error messages), then test each member of the list.

Equivalence class analysis

An equivalence class is a set of values for a variable that you consider equivalent.

Test cases are equivalent if

- (a) they test the same thing,
- (b) if one of them catches a bug, the others probably will too, and
- (c) if one of them doesn't catch a bug, the others probably won't either.

The set of values you could enter into a variable is the variable's domain. Equivalence class ***analysis*** divides a variable's domain into non-overlapping subsets (partitions) that contain equivalent values.

In equivalence-class ***testing***, you test one or two values from each partition.

Equivalence-class based testing makes testing more efficient by reducing redundancy of the tests.
As a coverage-oriented technique: test all the equivalence classes of every variable.

Boundary testing

In boundary-value testing, you partition the values of a variable into its equivalence classes and then test the upper and lower bounds of each equivalence class.

A boundary value is a particularly good member of an equivalence class to test because:

- It carries the same risks as all the other members of the class
- Boundary values carry an additional risk because off-by-one errors are common

Boundary-value testing adds a risk model to equivalence-class based testing.
Coverage: test every boundary of every variable.

Best representative testing

The best representative of a partition (of the domain of a variable) is the one most likely to cause the program to fail a test.

- If you can order values in the domain from small to large, best representatives are typically boundary values
- If you cannot order values, you can often find a best representative by considering more than one risk. Two values of a variable might be equivalent with respect to one risk, but not with respect to the other.
- If all values in a partition are truly equivalent, you can use any of them as a best representative.

The concept of "best representatives" generalizes domain testing to non-ordered sets and to secondary dimensions.
Coverage: test every best representative of every variable, relative to every risk.

Domain testing

Domain testing formalizes and generalizes equivalence class and boundary analysis:

- Partition the variable's domain into equivalence classes and test best representatives
- Test output domains as well as input domains
- Test secondary as well as primary dimensions
- Test consequences as well as input filters
- Test multidimensional variables and multiple variables together

After today, we'll stop talking about equivalence class and boundary analysis as techniques. They are all part of domain testing.

Test idea catalogs

You can develop a standard set of tests for a specific type of object (or risk) and reuse the set for similar things in this product and later products.

Marick (1994) suggested that testers develop these types of lists and called them test idea catalogs. See Marick (undated) for an updated set.

Kaner, Bach & Pettichord (2001, pp. 45-50) provide examples of test idea catalogs for numeric input variables, with more of the same type of catalogs in Kaner, Hoffman & Padmanabhan (2012). Hendrickson (2006), Hunter (2010), and Nguyen et al (2003) provide additional examples.

Coverage:
The catalog lists the test ideas to cover.

Multivariable testing

All-pairs testing is the best-known multivariable technique. It is effective for testing many **independent** variables.

Several classes of multivariable techniques:

- **Mechanical.** The tester uses a routine procedure to determine a good set of tests. Examples: random combinations and all-pairs.
- **Risk-based.** The tester combines test values (values of each variable) based on perceived risks associated with noteworthy combinations
- **Scenario-based.** The tester combines test values on the basis of interesting stories created for the combinations

All-pairs is defined by its coverage (all pairs of values of interest of all variables). The other approaches are coverage-focused to the extent that you design a pool of tests and attempt to cover it.

LOGICAL EXPRESSIONS

Consider a health-insurance program with a decision rule that says:

- if PERSON-AGE > 50 and
- if PERSON-SMOKES is TRUE
- then set OFFER-INSURANCE to FALSE.

The decision rule expresses a logical relationship.

If you make a series of separate decisions, the result is the same as if you had made all those decisions at the same time. Thus, you can test the set together as one complex logical expression.

Testers often represent decision rules (and combinations of rules) in decision tables. You can turn each row in the table into a test.

Amman & Offutt (2008) describe several coverage rules for logical-expression testing. Marick (2000) presents a simpler approach to coverage, which you can apply using MULTI (see the references).

As a coverage-oriented technique, logical-expression testing attempts to check every decision in the program or a theoretically interesting subset.

State-model-based testing

A program moves from state to state. In a given state, some inputs are valid and others are ignored or rejected.

In response to a valid input, the program under test does something that it can do, which takes it to a new state.

Think of a sequence of length 2 (from a state to a transition to the next state), length 3 (state → transition → state → transition → state), etc.

(Cross-reference: see the Operational Modes tour.)

Coverage:
State-model testers often use specialized algorithms to walk the program through long paths that cover all sequences of length 2.

User interface testing

- User interface testing is about checking that the elements of the UI have been implemented correctly.
- User interface testing is NOT about whether the UI is well designed or easy to understand or work with—***that's usability testing***

Coverage:
Focus on covering all the elements of the user interface (the dialogs, menus, pull-down lists, and all the other UI controls)

SPECIFICATION-BASED TESTING

Spec-based testing is focused on verifying factual claims made about the product in the specification. (A factual claim is any statement that can be shown to be true or false.)

This often includes claim made in the manual, in marketing documents or advertisements, and in technical support literature sent to customers.

We'll see in a few days that many specifications are implicit. For example, many programs do arithmetic but few include explicit specifications of the rules of arithmetic.

Coverage:
Test every claim in the documents that guide testing.

REQUIREMENTS-BASED TESTING

Requirements-based testing is focused on proving, requirement by requirement, that:

- the program satisfies every requirement in a requirements document, or that
- some of the requirements have not been met.

It might not be possible to answer: "Does this program actually **meet** this requirement" with simple tests. Requirements that are easily testable are often trivial compared to the "real" requirements.

What is called requirements-based testing is typically focused on written requirements. These are, of course, incomplete, subject to frequent change, and often incorrect.

Compliance-driven testing

Some products must meet externally-imposed requirements (such as regulatory requirements).

Compliance-driven testing is focused on doing the set of tasks (usually the minimum set) needed to demonstrate compliance with these requirements.

Coverage:
Do every task needed to demonstrate compliance.

CONFIGURATION COVERAGE

If you have to test compatibility with 100 printers, and you have tested with 10, you have achieved 10% printer coverage. More generally, configuration coverage is the percentage of configuration tests the program has passed compared to the number you plan to run.

Why call this a test technique? Testers focused on this coverage objective are likely to craft methods to make high volume configuration testing faster and easier. This optimization of the effort to achieve high coverage is the underlying technique.

Configuration coverage is the percentage of configuration tests the program has passed compared to the number you plan to run.

Localization testing

- When you adapt a program to run in a different language, a different country, or a different culture, you make a specific set of changes.
- Software publishers who will localize their software typically design the software to make localization easy. In such a case, you can
  - Create a list of the things that *can be* changed for localization
  - Test the list to see what was actually changed, whether the changes worked, and whether anything that should have been changed was not changed.

Coverage:
Test against a list of localization-related changes and risks

Tester-based techniques focus on who does the testing

- User testing
- Alpha testing
- Beta testing
- Bug bashes
- Subject-matter expert testing
- Paired testing
- Eat your own dogfood
- Localization testing

There's a mystique in designing a technique around the type of person who tests. However, what they will actually do may have little to do with what you imagine will happen.

User testing

- This testing is done by the types of people who would typically use your product.
- User testing might be done:
  - at any time during development,
  - at your site or at theirs,
  - in carefully directed exercises or at the user's discretion.
- Some types of user testing, such as task analyses, are more like joint exploration (involving at least one user and at least one member of your company's testing team) than like testing by one person.

Testers:
Users (ideally, representative of your market) or people who the company treats as surrogates for users.

Alpha testing

- This testing is done early in development, usually by the software development group (programmers and/or testers).
- "Alpha" is a milestone with different meanings at different companies. In the typical alpha period:
  - The program is stable and complete enough for some level of functional testing
  - But not yet stable enough for the beta milestone
- Alpha might start immediately after the first feature is finished (for example in Extreme Programming) or not until all features are "complete" (coded but probably not yet working).

Testers:
Typically programmers and in-house testers who work closely with the programmers.

Beta testing

- **Typical case**: External users run almost-finished software on their own computers. This testing starts at the "beta" milestone.
- **Design beta**: User representatives or subject matter experts assess the software's design.
- **Marketing beta**: Pre-release to potential large customers, typically later and more stable than at "beta" milestone.
- **Compatibility beta**: External users test the product's compatibility with their software or hardware, typically because they have software or hardware that the development group doesn't have. Ideally, this starts as soon as the software can be tested for compatibility because adapting the software can be difficult under these circumstances.

Testers:

Typically people external to the company (or at least external to the development group). Typically representative of the market or owners of market-relevant equipment.

Bug bashes

- In-house testing using anyone who is available (e.g. secretaries, programmers, tech support, maybe even some managers).
- A typical bug-bash lasts a half-day and is done when the software is close to being ready to release.
- Note: we're listing this technique as an example, not endorsing it. Some companies have found it useful for various reasons; others have not.
  - often an ineffective replacement for exploratory testing
  - often seen as more effective by non-testing managers than by the testers

Testers:
Typically employee nontesters or testers who aren't assigned to test this product

SUBJECT-MATTER EXPERT TESTING

- Give the product to an expert on some issues addressed by the software, and request feedback (bugs, criticisms, and compliments).
- The expert may or may not be someone you would expect to use the product--her value is her knowledge, not her representativeness of your market or her skill as a tester.
- When the expert pairs with a tester or programmer (serves as a live oracle), the staff gain a new level of training as a side effect of the testing process.

Tester:
Someone who is seen as highly knowledgeable about the product category or its risks

Paired testing

- Two testers (or a tester and a programmer) testing together:
  - May share one computer and trade control of it while they test.
  - Or test on their own machines, with dual-monitor systems (one placed for easy reading by the other tester) so that each tester can easily see what's on the other's screen.
  - Collaboration might involve on tester reading (specs, bug reports, etc.) or writing up a bug while the other executes tests
  - One tester might protect the other's time by dealing with all the visitors (e.g. manager nagging for status report).

Testers:
Two people (testers and/or programmers) on the project team, testing together.

EATING YOUR OWN DOGFOOD

- Your company uses and relies on pre-release versions of its own software.
- This often yields more critical design feedback than beta testing.
- This often provides a harsher and more credible real-world readiness assessment of the software than beta or formal in-house testing.

Testers:
In-house users who do real work with the software.
Caution: This can miss ways that other organizations will use the software. It might provide false reassurance about the quality of the software.

Localization testing

- The software is adapted to another culture or language.
- The localization testing is done by people from that culture or who are fluent in that language (probably a native speakers).
- These people are regarded as subject matter experts who can speak authoritatively about the appropriateness of the localization.

Testers:
People from (or deeply familiar with) the target culture.

Risk-based techniques focus on potential problems

- Boundary testing
- Quicktests
- Constraints
- Logical expressions
- Stress testing
- Load testing
- Performance testing
- History-based testing
- Risk-based multivariable testing
- Usability testing
- Configuration / compatibility testing
- Interoperability testing
- Long sequence regression

Risk-based testing starts from an idea of how the program could fail. Then design tests that try to expose problems of that type.

Boundary testing

Boundary testing arises out of a specific risk:

- Even if every other value in an equivalence class is treated correctly, the boundary value might be treated incorrectly (grouped with the wrong class)
  - The programmer might code the classification rule incorrectly
  - The specification might state the
  - The specifier might misunderstand the natural boundary in the real world
  - The exact boundary might be arbitrary, but coded inconsistently in different parts of the program.
- Example: Class should contain all values < 25 but 25 is treated as a member of the class as well.

Risk(s):
Misclassification of a boundary case or mishandling of an equivalence class.

QUICKTESTS (RISK-BASED TESTING)

A quicktest is an inexpensive test, optimized for a common type of software error, that requires little time or product-specific preparation or knowledge to perform. For example:

- **Boundary-value tests** check whether a variables boundaries were misspecified. You don't have to know much about the program to do this type of test.
- **Interference tests** interrupt the program while it's busy. For example, you might try cancelling a print job or forcing an out-of-paper condition while printing a long document.

We cataloged a lot of quicktests at the 7th Los Altos Workshop in Software Testing (1999) and will look at some of these in Lecture 2.

CONSTRAINTS

A constraint is a limit on what the program can handle. For example, if a program can only handle 32 (or fewer) digits in a variable, the programmer should provide protective routines to detect and reject any input outside of the 32-digit constraint.

Jorgensen (1999) and Whittaker (2002) provides detailed suggestions for identifying and testing:

- Input constraints
- Output constraints
- Computation constraints
- Stored-data constraints

Jorgensen & Whittaker's approach to constraints generalizes the idea of input boundaries to all program data and activities.

LOGICAL EXPRESSIONS

Consider an insurance program's decision rule:

- if PERSON-AGE > 50 and
- if PERSON-SMOKES is TRUE
- then set OFFER-INSURANCE to FALSE.

You can write this decision as a logical express (a formula that evaluates to TRUE or FALSE)

If you make a series of separate decisions, the result is the same as if you had made all those decisions at the same time. Thus, you can test the set together as one complex logical expression.

Marick (2000) took a risk-oriented approach to logical-expression testing by considering common mistakes in designing/coding a series of decisions. His approach is implemented in MULTI.

Stress testing

- Testing designed and intended to overwhelm the product, forcing it to fail.
  - Intentionally subject the program to too much input, too many messages, too many tasks, excessively complex calculations, too little memory, toxic data combinations, or even forced hardware failures.
  - Explore the behavior of the program as it fails and just after it failed.

(see for example, Beizer at http://www.faqs.org/faqs/software-eng/testing-faq/section-15.html)

- **What aspects of this program need hardening** to make consequences of failure less severe?

There are many other definitions of "stress testing," including what we are calling "performance testing" and "load testing."

LOAD TESTING

Load testing addresses the risk that a user (or group of users) can unexpectedly run the software or system under test out of resources.

A weak load test simply checks the number of users who can connect to a site or some equivalent count of obvious, simple task.

A better load testing strategy takes into account that different users do different tasks that require different resources. On a system that can handle thousands of connections, a few users doing disk-intensive tasks might have a huge impact.

Additionally, Savoia (2000) found that for many programs, as load increased, there was an exponential increase in the probability that the program would fail basic functional tasks.

Risks:

Inappropriate responses to high demands or low resources. Does the program handle its limitations gracefully or is it surprised by them? Does it fail only under extreme cases?

Performance testing

Testers usually run performance tests to determine how quickly the program runs (does tasks, processes data, etc.) under varying circumstances.

In addition, performance tests can expose errors in the software under test or the environment it is running on.

Run a performance test today; run the same test tomorrow:

- If the execution times differ significantly and
- the software was not intentionally optimized, then
- something fundamental has changed for (apparently) no good reason.

Risks:
Program runs too slowly, handles some specific tasks too slowly, or changes time characteristics because of a maintenance error.

History-based testing

These tests check for errors that have happened before.

- This includes studying the types of bugs that have occurred in past versions of this product or in other similar products. What's difficult for one product in a class is often difficult for other products in the same class. This isn't regression testing, it's history-informed exploration.
- In a company that has a regression problem (bugs come back after being fixed), regression tests for old bugs is a risk-based test technique.

Risks:
Old bugs reappear.

Risk-based multivariable testing

The most widely discussed multivariable techniques are mechanical (e.g. all-pairs testing). An algorithm determines what values of which variables to test together.

A risk-based technique selects values based on a theory of error.

- **Example**: you might test configurations (select video, printer, language, memory, etc.) based on troublesome past configurations (technical support complaints)
- **Example**: you might pick values of variables to use together in a calculation to maximize the opportunity for an overflow or a significant rounding error.

Risks:
Inappropriate interactions between variables (including configuration or system variables)

CONFIGURATION / COMPATIBILITY TESTING

Software runs in an environment (the computer or network that it runs on).

Its environmental requirements might be:

- narrow (only *this* operating system, *that* printer, at least *this much* memory, works only with *this version* of that program), or
- very flexible.

Configuration tests help you determine what environments the software will *correctly* work with.

Configuration testers often pick specific devices, or specific test parameters, that have a history of causing trouble.

Risk:
Incompatibility with hardware, software, or the system environment.

INTEROPERABILITY TESTING

- Test whether the software under test interacts correctly with another program, device, or external system.
- Simple interoperability testing is like function testing: Try the two together. Do they behave well together?
- To add depth to this testing, you can design tests that focus specifically on ways in which you suspect the software might not work correctly with the other program, device or system.
- A tester focused on interoperability will probably test from a list of common problems.

Difference between compatibility testing and interoperability testing:

- **Compatibility—with software or hardware that are part of the system under test**
- **Interoperability—with software or hardware external to the system under test**

Usability testing

Usability tests try to demonstrate that some aspect of the software is unusable for some members of the intended user community. For example:

- Too hard to learn
- Too hard to use
- Makes user errors too likely
- Wastes your time

Usability testing: done by usability testers who might or might not be end users.

User testing: done by users, who may or may not focus on the usability of the software.

Risks:
Software is unusable for some members of the intended user community. (e.g. too hard to learn or use, too slow, annoying, triggers user errors, etc..)

Long-sequence regression

A program passes a set of tests. Then test the same build of the same software with the same tests, run many times in a random order. This is long-sequence regression testing.

Long-sequence regression can expose bugs that are otherwise hard to find, such as intermittent-seeming failures from:

- memory leaks,
- race conditions,
- wild pointers and
- other corruption of working memory or the stack.

See McGee & Kaner (2004).

The long-sequence tests hunt bugs that won't show up in traditional testing (run tests one at a time and clean up after each test) and are hard to detect with source code analysis.

Activity-based techniques focus on how you do the testing

- Guerilla testing
- All-pairs testing
- Random testing
- Use cases
- Scenario testing
- Installation testing
- Regression testing
- Long sequence testing
- Dumb monkey testing
- Load testing
- Performance testing

Because these focus on "how-to", these might be the techniques that most closely match the classical notion of a "technique."

GUERILLA TESTING

- Exploratory tests that are usually time-boxed and done by an experienced explorer: The goal is a fast and vicious attack on some part of the program.
- For example, a senior tester might spend a day testing an area that is seen as low priority and would otherwise be ignored. She tries out her most powerful tests.
  - If she finds significant problems, the area will be rebudgeted and the overall test plan might be affected.
  - If she finds no significant problems, the area will hereinafter be ignored or only lightly tested.

Activity:

Time-boxed, risk-based testing focused on one part of the program.

All-pairs testing

Suppose you test N variables together. Pick a few values to test from each variable.

Under the all-pairs coverage criterion, your tests must include one value for each variable and, across the set of tests, every value of each variable is paired with every value of every other variable. (Cohen et al, 1996).

Testers typically rely on a tool for picking the combinations of values for the variables. See http://www.pairwise.org/tools.asp

All-pairs specifies a coverage criterion.

Activity: following the algorithms (or using tools) to generate tests that meet this criterion.

RANDOM TESTING

In **random testing**, the tester uses a random number generator to determine:

- The values to be assigned to some variables, or
- The order in which tests will be run, or
- The selection of features to be included in a this test.

Activity:

Drive test decisions with a random number generator.

Use cases

A **use case** describes a system's behavior in response to a request from an *actor* which might be a human or another system.

The use case describes intended behavior (how the system should work) but not the motivation of the actor or the consequences for the actor if the request fails.

The use case shows the actor's steps and system behavior on a sequence diagram.

The diagram's "happy path" shows the simplest set of steps that lead to success. Other paths show complications, some leading to failures.

In use-case based testing, you do the modeling and test down the sequence diagram's paths.

See Utting & Legeard (2007).

Activity:
The tester creates sequence diagrams (behavior models) and runs tests that trace down the paths of the diagrams.

SCENARIO TESTING

A scenario is a hypothetical story about the software. A **scenario test** is a test based on a scenario. A good scenario test has five characteristics:

1. The scenario is a coherent story
2. The story is credible
3. Failure of the test would motivate a stakeholder with influence to argue that it should be fixed
4. The test is complex (involves several features or data)
5. The test result is easy to evaluate.

Activity:
Creating a story (or a related-family of stories) and a test that expresses it.

Installation testing

- Check whether a new product, a new version of the product, or a patch installs well, without interfering with other software.
- Check whether a virgin installation or re-installation works.
- Check whether uninstallation works and reinstallation after uninstallation is possible (or impossible if it the Digital Rights Management (DRM) forbids it.)
- Installation is often one of the least well-tested parts of a program, and therefore a good place to hunt bugs.

Some discussions of installation testing are risk-focused (such as Bach, 1999), but many are more procedural, or more focused on how to automate much of the installation-test process (e.g. Agruss, 2000).

Regression testing

Reuse of the same tests after change.

- The goal of *bug fix regression* is to prove a bug fix was ineffective.
- The goal of *old bugs regression* is to prove a software change caused a fixed bug to become unfixed.
- The goal of *side-effect regression* is to prove a change has caused something that used to work to now be broken.

Most discussions of regression testing as a technique consider:

- How to automate the tests, or
- How to use tools to select a subset of the tests that might be the most interesting for the current build.

Activity:

Do the same boring tests over and over.

Or write and fix and fix and fix and fix and fix and fix and fix "automation" code to do the same test over and over.

Long sequence testing

Long sequence testing (LST) is done overnight or for days. Long-sequence regression is 1 example. You can also create long sequences of tests that have never been run previously.

- The goal of LST is to discover errors that short sequence tests miss.
- These are often basic functional errors that are unnoticed when they occur but show up as violations of a precondition for a later test.
- Other errors include , stack overflows, wild pointers, memory leaks, and bad interactions among several features.
- LST is sometimes called duration testing, life testing, reliability testing, soak testing or endurance testing.

Activity:
Creating software to execute LSTs, with diagnostics to help troubleshoot the failures they trigger.

Dumb monkey testing

When a program is in any given state, it will ignore some inputs (or other events) and respond to others. The program's response takes it to its next state. This is a state transition.

You can feed random inputs to the program to force state transitions. Noel Nyman calls this "monkey" testing.

If you have

- a state model that ties inputs to transitions, and
- an oracle (the ability to tell whether the program transitioned to the correct state)

then you can do state-model-based testing. (Nyman calls this a "smart monkey").

If you don't have the oracle, you can still run the monkey, waiting to see if the program crashes or corrupts data in some obvious way. This is the dumb monkey.

Activity:

Random state-transition tests programmed to run-until-crash.

Performance testing

"Performance testing is a type of testing intended to determine the responsiveness, throughput, reliability, and/or scalability of a system under a given workload…

"Performance testing is typically done to help identify bottlenecks in a system, establish a baseline for future testing, support a performance tuning effort, determine compliance with performance goals and requirements, and/or collect other performance-related data to help stakeholders make informed decisions related to the overall quality of the application being tested." (Meier et al., 2007, 3-4).

Activity:

Code and execute input streams and execution-timing monitors.

Evaluation-based techniques focus on your oracle

- Function equivalence testing
- Mathematical oracle
- Constraint checks
- Self-verifying data
- Comparison with saved results
- Comparison with specifications or other authoritative documents
- Diagnostics-based testing
- Verifiable state models

Any time you have a well-specified oracle, you can build a set of tests around that oracle.
See our presentation of Hoffman's collection of oracles in BBST-Foundations.

Function equivalence testing

"Function equivalence tests compare two programs' evaluation of the same … function." (Kaner et al., 1993, p.135).

- The function in the software under test is the test function; the other is the reference function or the oracle function.
- In this type of testing, you might compare the program's evaluations of hundreds (or billions) of sets of data. At some point, you either find a difference between the functions or conclude you have tested so much that you won't find a difference with further testing.

Example: Hoffman's (2003) on testing MASPAR's square root function.

Mathematical oracle

- You can often derive a predicted value from the mathematical attributes of the software under test. For example:
  - invert calculations (square a square root, or invert a matrix inversion)
  - a sine function's shape is predictable
- This is like function equivalence testing
  - You might test with arbitrarily many values
  - You make and check exact predictions
  - **But** you might do everything within the software under test, without an external reference function.

Constraint checks

Check whether some output of the program is impossible:

- An American postal code can't be 6 digits
- A Canadian province's name can be checked against a short list of provinces
- In an order entry system, the order number of an order should be smaller than the number of an order that was placed later.

Something doesn't have to be truly impossible. It just has to be unlikely enough that it would be worth your time to investigate why the program gave that result.

SELF-VERIFYING DATA

Embed the correct test result in a set of test data. For example:

- Add a comment field to a database of test case records
- Provide a checksum, hash code, or digital signature to authenticate the result

Similarly, you could build functions into the program under test that serve as the equivalent of embedded test data by providing the should-be-correct result on demand.

Comparison with saved results

Regression testing is the most common example of a technique built around saved results.

- Run a test
- If the program passes, keep its output data
- After a new build, run the test again
- Check whether the new test results match the saved test results.

Comparison with specifications or other authoritative documents

Specification-based testing checks the product against every factual claim made about the product in the specification or any other document that the program must verify against. (A factual claim is any statement that can be shown to be true or false.)

- When you think about **every** claim, you are thinking about specification-based testing in terms of coverage.
- When you think about **what** claims, you are treating the specification as an oracle.

DIAGNOSTICS-BASED TESTING

- Run a test
  - As part of the normal (or test-customized) operation of the program, the program runs diagnostics. If the test triggers an unusual state, the program reports a diagnostic issue.

  OR

  - The tester runs a diagnostic immediately after running the test.
- The diagnostics can expose effects of the test that would otherwise be invisible, such as memory corruption, assignment of incorrect values to internal variables, tasks that were only half-completed, etc.

An oracle is a mechanism or heuristic principle for determining whether a program has a problem. Here, the diagnostics are the mechanism. They might not tell you what the "right" behavior is; they alert you that something looks wrong.

Verifiable state models

You have an oracle whenever you can compare your program's behavior to a model of how it should behave.

When a program is in any given state, it will ignore some inputs (or other events) and respond to others. The program's response takes it to its next state. This is a state transition.

You can do state-model-based testing if you have:

- a state model that ties inputs to transitions, and
- ability to tell whether the program is actually in the state predicted by the model.

We emphasize the oracle aspect of state testing to the extent that we can make a detailed comparison between the expected state and the test-result state.

Desired-result techniques focus on a specific decision or document

- Build verification
- Confirmation testing
- User acceptance testing
- Certification testing

You are doing document-focused testing if you run a set of tests primarily to collect data needed to fill out a form or create a clearly-structured report.

Build verification

It would waste your time to test a build that had problems like these:

- Missing critical features or files
- Built (accidentally) with an outdated version of some module(s)
- Bugs that significantly destabilize the version.

Many groups follow the rule that if the program fails ANY build verification tests, the build is sent back to the programmers without further testing.

The suite of BVTs is typically automated, and contains a relatively small number of tests.

BVT is focused around a desired result: Determine whether the build is complete enough and stable enough to warrant more thorough testing.

CONFIRMATION TESTING

Test groups might run a carefully designed suite of confirmation tests when their company is required to demonstrate that the program has certain characteristics or operates in a certain way.

For example, some contracts for custom software provide for a user acceptance test and set detailed expectations about the testing. The testers might create a suite of demonstrations that the program meets these expectations. (These tests might or might not be the actual suite used by the customer for acceptance testing.)

User acceptance testing

In early times, most software development was done under contract. A customer (e.g. the government) hired a contractor (e.g. IBM) to write a program. The customer and contractor would negotiate the contract. Eventually the contractor would say that the software is done and the customer or her agent (such as an independent test lab) would perform acceptance testing to determine whether the software should be accepted.

If software failed the tests, it was unacceptable and the customer would refuse to pay for it until the software was made to conform to the promises in the contract (which were what was checked by the acceptance tests).

This is the same meaning we adopted in Foundations. As we noted then, there are many other definitions of "acceptance testing."

CERTIFICATION TESTING

The test group might be required to **certify** (attest in writing) that the product has specific characteristics. For example:

- Certify compliance with IEEE Standard for Floating-Point Arithmetic (Std 754).
- Certify that all classes of the code were inspected
- Certify the software was tested to a level of 100% statement-and-branch coverage.

The test group does whatever tasks are needed to be able to honestly make the required certification. (It may be as simple as running a standard certification suite.) To the extent that these tasks include testing, they may not look like **good** testing. The test group will probably do the minimum necessary for the certification. Narrowing the focus is part of the technique.

Review

- You should know what function tests are and how to tour a program to find most of its functions.
- When someone describes a technique to you, you should be able to figure out its scope and whether it is focused mainly on
  - coverage,
  - risk,
  - who does the testing,
  - how to do the test,
  - how to evaluate the test, or on
  - certifying the program meets a specific criterion
- You should be able to imagine relying on that technique but changing it (or using other techniques) to strengthen the areas that are out of focus (e.g. improve coverage or be more sensitive to risk or adapt the technique for end users.)

End of Lecture 1

Black Box Software Testing: Introduction to Test Design: Lecture 2: Risk-Based Testing

Cem Kaner, J.D., Ph.D.
Professor of Software Engineering: Florida Tech

Rebecca L. Fiedler, M.B.A., Ph.D.
President: Kaner, Fiedler & Associates

These notes are partially based on research that was supported by NSF Grants EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers" and CCLI-0717613 "Adaptation & Implementation of an Activity-Based Online or Hybrid Course in Software Testing." 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.

OVERVIEW OF THE COURSE

	Technique	Context / Evaluation
1	Function testing & tours.	A taxonomy of test techniques.
2	Risk-based testing, failure mode analysis and quicktests	Testing strategy. Introducing the Heuristic Test Strategy Model.
3	Specification-based testing.	(… work on your assignment …)
4	Use cases and scenarios.	Comparatively evaluating techniques.
5	Domain testing: traditional and risk-based	When you enter data, any part of the program that uses that data is a risk. Are you designing for that?
6	Testing combinations of independent and interacting variables.	Combinatorial, scenario-based, risk-based and logical-implication analyses of multiple variables.

REFERENCES: REQUIRED READINGS ARE IN BOLDFACE

Quicktests

- **Hendrickson, E. (2006). Test heuristics cheat sheet. http://testobsessed.com/wp-content/uploads/2011/04/testheuristicscheatsheetv1.pdf**
- Edgren, R. (2011). The Little Black Book on Test Design. http://thetesteye.com/blog/2011/09/the-little-black-book-on-test-design/
- Hunter, M. J. (2010). You are not done yet. http://www.thebraidytester.com/downloads/YouAreNotDoneYet.pdf
- Kaner, C. & Johnson, B. (1999) Styles of exploration, 7th Los Altos Workshop on Software Testing. http://www.kaner.com/pdfs/LAWST7StylesOfExploration.pdf
- Whittaker, J.A. (2002) How to Break Software, Addison Wesley

Guidewords

- **Bach, J. (2006). Heuristic test strategy model, Version 4.8. http://www.satisfice.com/tools/satisfice-tsm-4p.pdf**
- HAZOP Guidelines (2011). Hazardous Industry Planning Advisory Paper No. 8, NSW Government Department of Planning. http://www.planning.nsw.gov.au/LinkClick.aspx?fileticket=HT4S8cZpZ3Y%3d&tabid=168&language=en-AU

Failure modes

- Kaner, C. (1988), Testing Computer Software, presented a list of 480 common software problems. http://logigear.com/articles-by-logigear-staff/445-common-software-errors.html
- **Vijayaraghavan, G., & Kaner, C.(2003). Bug taxonomies: Use them to generate better tests. Software Testing Analysis & Review Conference. http://www.testingeducation.org/a/bugtax.pdf**

TEST DESIGN

In Lecture 1, we

- studied two related techniques (touring and function testing)
- raced through a zillion other techniques
- studied seven common attributes of test techniques.

Test strategy

Today, we look at a few concepts important for developing a testing strategy:

- Test cases
- Comparing test techniques in terms of their strengths and blind spots
- Context factors that influence test strategy
- Information objectives that drive test strategy

Context and information objectives are (or should be) the drivers of any testing strategy.

What's a test case?

Should your designs focus on procedure?

- "A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement." (IEEE)

Focus on the test idea?

- "A test idea is a brief statement of something that should be tested. For example, if you're testing a square root function, one idea for a test would be 'test a number less than zero'. The idea is to check if the code handles an error case."

(Marick, http://www.exampler.com/testing-com/tools.html)

Test cases

We're more interested in the informational goal.

The point of running the test is to gain information, for example whether the program will pass or fail the test.

A test case

is a question

you ask the program.

Testing strategy

Your testing strategy is

- the guiding framework for deciding what tests (what test techniques) are best suited to your project,
- given your project's objectives and constraints (your context)
- and the informational objectives of your testing.

See Bach's "Heuristic Test Planning: Context Model" http://www.satisfice.com/tools/satisfice-cm.pdf

ATTRIBUTES OF "GOOD" TESTS (MORE ON THIS IN LECTURE 4)

- Power
- Valid
- Value
- Credible
- Representative
- Non-redundant
- Motivating
- Performable
- Reusable
- Maintainable
- Information value
- Coverage
- Easy to evaluate
- Supports troubleshooting
- Appropriately complex
- Accountable
- Affordable
- Opportunity Cost

Most tests have these attributes to some degree. To evaluate a test, imagine possible tests that would have more of the attribute or less of it. Compared to those, where does this one stand?

EVERYONE TESTS IN A CONTEXT

- Harsh constraints
  - Complete testing is impossible
  - Finite project schedules and budget
  - Limited skills of the test group
- You might do your testing before, during or after the product under test is released.
- Improvement of product or process might or might not be an objective of testing.
- You test on behalf of stakeholders
  - Project manager, marketer, customer, programmer, competitor, attorney
  - Which stakeholder(s) this time?
    - What information are they interested in?
    - What risks do they want to mitigate?

As service providers, it is our task to learn (or figure out) what services our clients want or need *this time*, and *under these circumstances*

Examples of context factors that drive and constrain testing
These differ from project to project

- Who are the stakeholders with influence?
- Are there non-stakeholders with influence (e.g. regulators)?
- What are the goals and quality criteria for the project?
- What skills and resources (such as time, money, tools, data, technology and testability support) are available?
- What's in the product?
- How could it fail?
- Potential consequences of potential failures?
- Who might care about which consequence of what failure?
- How to recognize failure?
- How to decide what result variables to attend to?
- How to decide what *other* result variables to attend to in the event of intermittent failure?
- How to troubleshoot and simplify a failure, so as to better
  - motivate a stakeholder who might advocate for a fix?
  - enable a fixer to identify and stomp the bug more quickly?
- How to expose, and who to expose to, undelivered benefits, unsatisfied implications, traps, and missed opportunities?

Context: Imagine 2 companies making similarly-capable products for the same market.

PROJECT 1 Resources	PROJECT 2 Resources
• Mature product. • Lots of automated GUI regression test code, created in previous versions. • Some testers have good programming skills and know the regression tool's language. • Time available in the schedule for a thorough round of regression test code maintenance.	• New product. Tight schedule. • No pre-existing tests. • Testers know the subject matter, the product environment, and some are excellent bug hunters. • None of the testers are skilled programmers.
If I was managing this project, I would probably plan for a lot of automated GUI regression testing.	**If I was managing this project, I'd probably plan for intensely exploratory testing: Risk-focused, no automated regression, not much test documentation.**

Testing strategy in context

Your goal should NOT be to impose "best practices" or "standards" on your client.

Your goal should be to help your client do the best that it can ***under the circumstances.***

**Sometimes,
doing the best you can
under the
circumstances
includes
changing the
circumstances.**

Common information objectives

- Find important bugs
- Assess the quality of the product
- Help managers make release decisions
- Block premature product releases
- Help predict and control product support costs
- Check interoperability with other products
- Find safe scenarios for use of the product
- Assess conformance to specifications
- Certify the product meets a particular standard
- Ensure the testing process meets accountability standards
- Minimize the risk of safety-related lawsuits
- Help clients improve product quality & testability
- Help clients improve their processes
- Evaluate the product for a third party

Different objectives require different testing tools and strategies and will yield different tests, test documentation and test results.

STRATEGY AND DESIGN

Think of the design task as applying the strategy to the choosing of specific test techniques and generating test ideas and supporting data, code or procedures:

- Who's going to run these tests? (What are their skills / knowledge)?
- What kinds of potential problems are they looking for?
- How will they recognize suspicious behavior or "clear" failure? (Oracles?)
- What aspects of the software are they testing? (What are they ignoring?)
- How will they recognize that they have done enough of this type of testing?
- How are they going to test? (What are they actually going to do?)
- What tools will they use to create or run or assess these tests? (Do they have to create any of these tools?)
- What is their source of test data? (Why is this a good source? What makes these data suitable?)
- Will they create documentation or data archives to help organize their work or to guide the work of future testers?
- What are the outputs of these activities? (Reports? Logs? Archives? Code?)
- What aspects of the project context will make it hard to do this work?

Techniques and strategy

We can't teach you enough about design in this course to make you effective at developing a test strategy for a complex product.

What we hope to do is teach you

- enough
- about enough techniques
- for you to understand how much flexibility is available to you
- for tailoring your testing activities
- to your information needs
- in your context.

Now let's look at another technique (or family of techniques):

Risk-Based Testing

Risk

The possibility of suffering harm or loss

In software testing, we think of risk on three dimensions:

- How 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 concern is:

- **how the product can fail.**

For project management:

- **how likely**
- **what consequences**

RISK-BASED TESTING

For the test designer, the essence of risk-based testing is:

- Imagine how the product can fail
- Design tests to expose these (potential) failures.

DIFFERENT APPROACHES TO RISK

1. Quicktests
2. Exploratory Guidewords
3. Failure Mode & Effects Analysis
4. Project-level risks

This list of quicktests evolved out of a bug taxonomy (Testing Computer Software) created to give testers ideas for testing common bugs. We extended it in presentations to the 7th Los Altos Workshop on Software Testing (Exploratory Testing, July 1999: Chris Agruss, Jim Bampos, Jack Falk, Dave Gelperin, Elisabeth Hendrickson, Doug Hoffman, III, Bob Johnson, Cem Kaner, Brian Lawrence, Hung Nguyen, Noel Nyman, Jeff Payne, Bret Pettichord, Drew Pritsker, Harry Robinson, Melora Svoboda, James Tierney, & Rodney Wilson). Several additions come from Bach & Bolton's Rapid Software Testing course.

Another useful collection of quicktest ideas, in an very interesting structure, was developed in Alan Jorgensen (1999)'s dissertation. Whittaker (2002) presented an expansion of this work, with more examples.

QUICKTESTS?

- A **quicktest** is
  - inexpensive,
  - easy to design, and
  - requires little knowledge, preparation or time to perform.
- Underlying every quicktest is a theory of error.

  If an error is so common that you are likely to see it

  » in many applications,

  » on several platforms

  you can develop a test technique optimized for that type of error.

Use quicktests because they are effective.

If a type of test doesn't expose many bugs in your environment, use something else.

Classic Quicktest: Shoe Test

Move the cursor to an input field. Put your shoe on the keyboard. Go to lunch.

Basically, you're using the auto-repeat on the keyboard for a cheap stress test.

- **This was one of the first tests for input buffer overflows.**
- **It was an effective test for a remarkably long time.**

This is a trivially simple introductory example. With all the valuable ideas for quicktesting that follow, it is disappointing when a student relies on this as an example in an exam…

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.

Intended domain:

$0 < X < 100$

Same as $1 \leq X \leq 99$

Common coding errors

$0 \leq X$ (accepts 0)

$X \leq 100$ (accepts 100)

$1 < X$ (rejects 1)

$X < 99$ (rejects 99)

Why are quicktests black box?

Simple boundary errors could be easily exposed by code inspection. So are many other types of bugs exposed by quicktests.

The obvious question:

Why run quicktests instead of doing more thorough code inspection?

Our answer:

As testers, we test the code that we get.

- If you routinely find certain types of errors, you should design tests that are optimized to find these types of errors cheaply, quickly, and without requiring tremendous skill.

COMMON IDEAS FOR QUICKTESTS

- User interface design errors
- Boundaries
- Overflow
- Calculations and operations
- Initial states
- Modified values
- Control flow
- Sequences
- Messages
- Timing and race conditions
- Interference tests
- Error handling
- Failure handling
- File system
- Load and stress
- Configuration
- Multivariable relationships

This is a convenient way to categorize a lot of quicktests, but it's not an authoritative structure. You can sort the same tests in many ways.

User interface design errors

Tour the user interface for things that are confusing, unappealing, time-wasting or inconsistent with relevant design norms.

Examples of test ideas:

- Check for conformity to Apple's Human Interface Guidelines
- Read menus, help and other onscreen instructions
- Try out the features
- Watch the display as you move text or graphics
- Force user errors. Intentionally misinterpret instructions, do something "foolish" and see what happens.

BOUNDARIES

The program expects variables to stick within a range of permissible values.

Examples of test ideas:

- Try inputs that:
  - are too big or too small
  - are too short or too long
  - create an out-of-bounds calculation
  - combine to create out-of-bounds output
    - can't be stored
    - can't be displayed
    - can't be passed to external app

OVERFLOW OR UNDERFLOW

These values are far too large or too small for the program to handle.

Examples of test ideas:

- Input empty fields or 0's
- Paste huge string into an input field
- Calculation overflows (individual inputs are OK but an operation (add, multiply, string concatenate) yields a value too big for the data type, e.g. integer overflow) or for a result variable that will store or display the result
- Read / write a file with too many elements (e.g. overflow a list or array)

"Overflow" values are too big for the program to process or store. A value can be out of bounds but not be so big that it causes an overflow.

Invalid calculations & operations

Calculation involves evaluation of expressions, like 5*2. Some expressions evaluate to impossible results. Others can't be evaluated because the operators are invalid for the type of data.

Examples of test ideas:

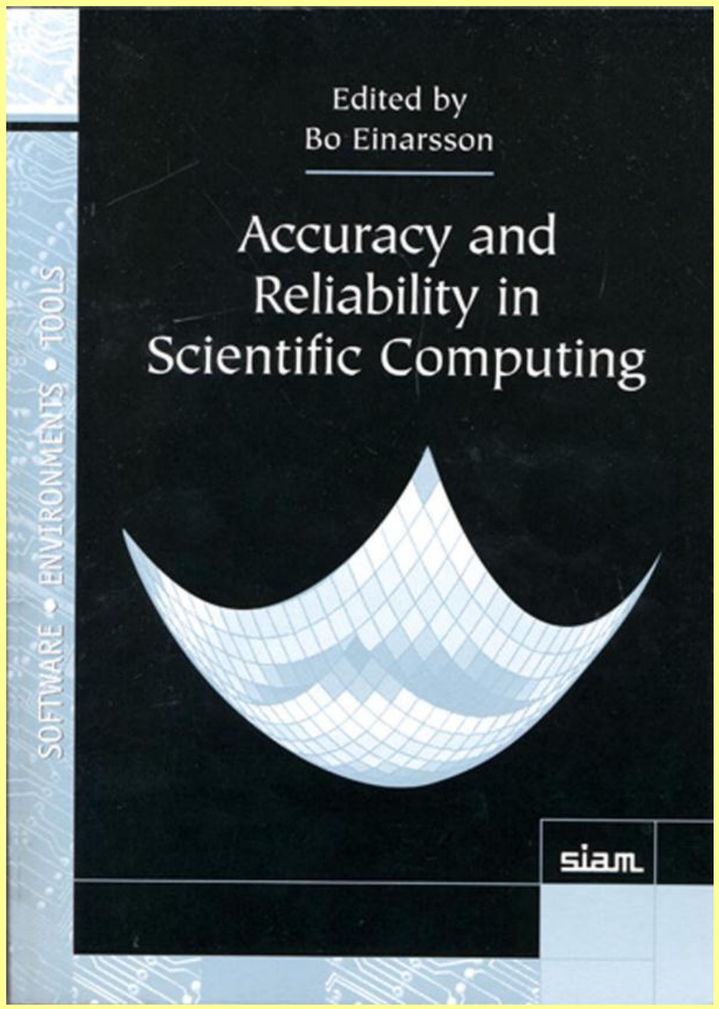
- Enter data of the wrong type (e.g. non-numbers into a numeric field)
- Force a divide by zero
- Force a divide by near-zero
- Arithmetic operations on strings
- String operations on numbers
- Arithmetic involving multiple numeric types: If you get a result, is it the type you expect?

Most errors that create a risk of invalid operations or impossible calculations are either caught at compile time or are more easily visible to a reader of the code.

Invalid calculations & operations

Caution:

- Some people think they can check calculations primarily with quicktests.
- Many calculations involve several variables that all have to be set carefully to achieve a powerful test.
- Because you have to know what you're doing, calculation tests are often **not** quicktests.



Initial states

What value does a variable hold the first time you use it? A variable might be:

- **Uninitialized** (not explicitly set to any value; holds random bits)
- **Initialized** (set to starting value, often 0; often given default value later)
- **Default** (set to a **meaningful** starting value)
- **Assigned or calculated** (intentionally set to a value appropriate to current need)
- **Carried over** (brings a previously assigned value to a new calculation that might expect the default value.)

A variable can be in any one of these 5 states. You have a bug if the program operates on the assumption that the variable is in one of the other states.

Initial states examples

1) Start with a fresh copy of the program (no saved data). Enter data into one dialog. Then do an operation (calculation or save) that uses data that you explicitly entered and data that have not been entered (you have not done any operation that would display those data).

   Are the unassigned data:

   – Uninitialized?

   – Initialized but to inappropriate values?

   – Default values?

2) Start with a variable that has a reasonable value. Enter an impossible value and try to save it, or erase the value. Does the program insert the old value? Default value? Something else?

We explored many initial state bugs in *Testing Computer Software.* Whittaker also illustrates several throughout *How to Break Software.*

Modified values

Set a variable to a value; then change it. This creates a risk if some *other* part of the program depends on this variable.

Examples of test ideas:

- In a program that calculates sales tax, buy something, calculate the tax, then change the person's state of residence.
- Change the location (address) of a device (make the change outside the program under test)
- Specify the parameters for a container of data (e.g. a frame that displays data, or an array that holds a number of elements). Then increase the amount of data. If there is auto-resize, increase and then decrease several times.

CONTROL FLOW

The control flow of a program describes what it will do next. A control flow error occurs when the program does the wrong next thing.

Example of a test idea:

A jump table associates an address with each event in a list. (An event might be a specific error or pressing a specific key, etc.) Press that key, jump to (or through the pointer in) the associated address.

When the program changes state, it updates the addresses, so the same actions do new things. If it updates the list incompletely or incorrectly, some new responses will be wrong. Table-driven programming errors are often missed by tests focused on structural coverage.

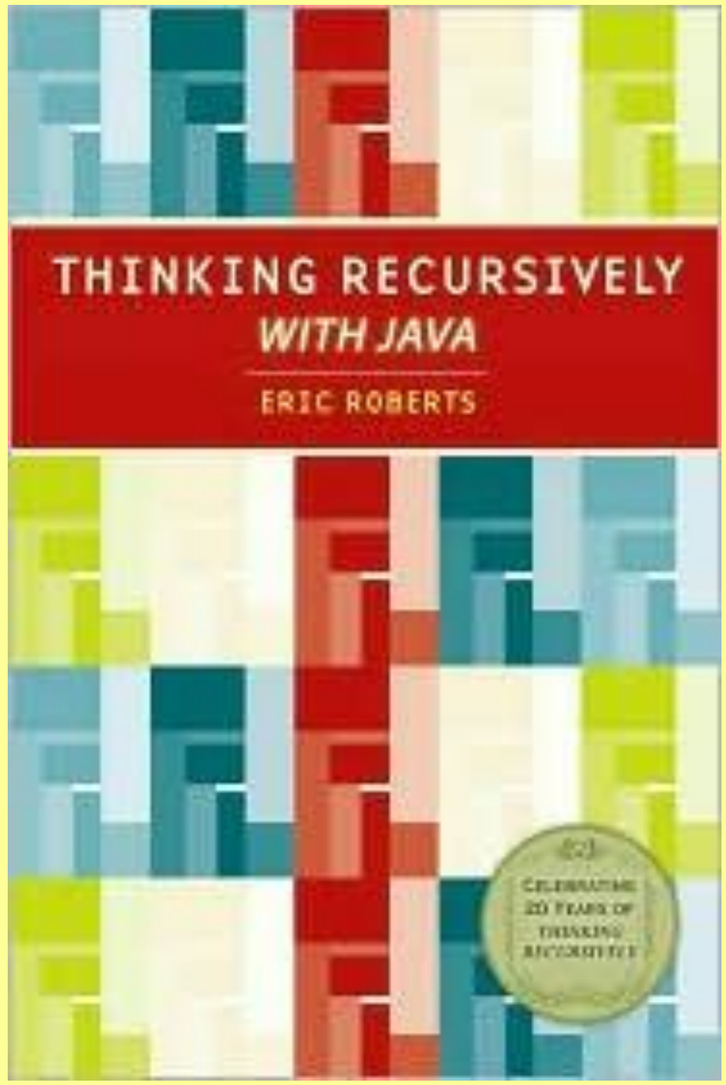
If the programmers achieved a high level of structural coverage in their testing, the control flow bugs that are left are usually triggered by special data values, interrupts or exceptions, race conditions or memory corruption.

SEQUENCES

A program might pass a simple test but fail the same test embedded in a longer sequence.

Examples of test ideas:

- Repeat the same test many times. Especially good targets:
  - Anything that creates an error message
  - Anything that halts a task in the middle. Exception-handler may not free up memory or reset variables the program was in the middle of calculating.
  - Anything that makes the program call itself (recursion). (Will this terminate? Will it exhaust system resources before terminating?)
- Run a suite of automated regression tests in a long randomized sequence.



Messages

If the program communicates with an external server or system, corrupt the messages between them.

Examples of test ideas:

- Corrupt the connection string. Some programs have a configuration file that includes a connection string to a remote resource, such as the database. What if the string is a little wrong? Can the program gain access anyway? How will it function without access?
- Program A sends a message to B, expecting a response. Normally, B will report success or failure. Corrupt the response so that it has elements of both (success and failure) and see which (if either) Program A believes.
- Corrupt the response so that it contains huge strings. Will this overflow a buffer or overwhelm Program A's error processing? (see Jorgensen, 2003 on hostile data streams).

TIMING, INCLUDING RACE CONDITIONS

Timing failures can happen if the program needs access to a resource by a certain time, or must complete a task by a certain time. This is a race condition if the program expects event A before B but gets B first.

Examples of test ideas:

- When providing input to a remote computer, don't complete entry until just before, just as, or just after the application times out (stops listening for your input).
- Delay input from a peripheral by making it busy, paused, or unavailable.

Interference tests

In interference testing, you do something to interfere with a task in progress. This might cause a timeout or a failed race condition. Or the program might lose data in transmission to/from an external system or device.

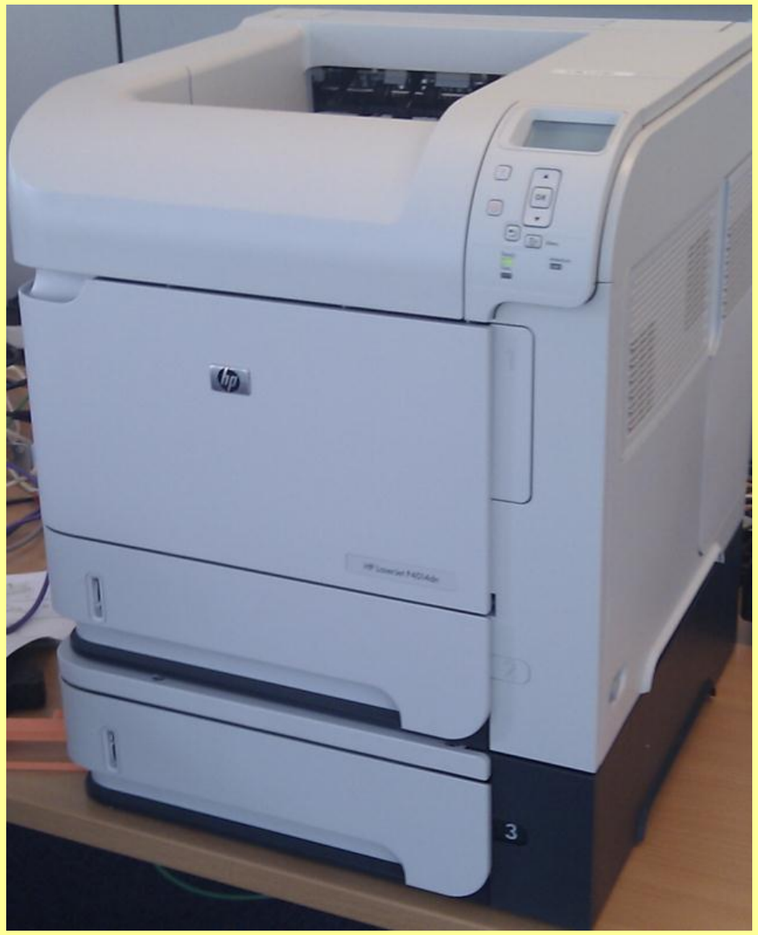
Examples of test ideas:

- Create interrupts
- Change something the task depends on
- Cancel a task in progress
- Pause a task in progress
- Compete for a resource needed by the task
- Swap task-related code or data out of memory.

INTERFERENCE TESTS: INTERRUPTS

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



From Jemimus, http://www.flickr.com/photos/jemimus/4559794553

Interference tests: Change

Change something this task depends on

- swap out a disk
- Disconnect/reconnect with a new IP address
- Disconnect/reconnect with a new router that uses different security settings
- 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

Interference tests: 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

Interference tests: 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
  - Sleep the computer
  - Put a database under use by a competing program, lock a record so that it can't be accessed — yet.

INTERFERENCE TESTS: SWAP

Swap a 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 (whatever the timeout period is). Does it come back? What's its state? What's 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.

A "process" is a program, typically one that is running now, concurrently with other programs (processes).

Interference tests: 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

Error handling

Errors in dealing with errors are among the most common bugs. These include:

- failure to anticipate the possibility of errors and protect against them
- failure to notice error conditions, and
- failure to deal with detected errors in a reasonable way.

Examples of test ideas:

- Make the program generate every error message. If two errors yield the same message, create both.
- After eliciting an error message, repeat the error several times. Check for a memory leak.
- After eliciting an error message, keep testing. Look for side effects of the error.

Failure handling

After you find a bug, you can look for related bugs.

Examples of test ideas:

- Keep testing after the failure. What vulnerabilities does recovery from the failure expose? (For example, data might not be properly saved after an exception-handling exit from a task.)
- Test for related bugs while troubleshooting this failure. Look for more serious or different symptoms by varying the test conditions.
- Test for related bugs after this bug was fixed.

File-system

Read or write to files under conditions that should cause a failure. How does the program recover from the failure?

Examples of test ideas:

- Read or write:
  - To a nonexistent file
  - To a locked (read-only) file
  - To a file that's open in another process (maybe another instance of this process) but not locked
  - To a file when you have insufficient privileges
  - To a file that exceeds the maximum file size
  - To a file that will overfill the disk (when writing) or memory (when reading)
  - To a disk with bad sectors
  - To a remote drive that is not connected
  - To a drive that is disconnected during the read or write

Load

Significant background activity eats resources and adds delays. This can yield failures that would not show up on a quiet system.

Examples of test ideas:

- Test (generally) on a significantly busy system.
- Run several instances of the same application in parallel. Open the same files.
- Try to get the application to do several tasks in parallel.
- Send the application significant amounts of input from other processes.

CONFIGURATION PROBLEMS

Check the application's compatibility with different system configurations:

- Progressively lower memory and other resources until the product gracefully degrades or ungracefully collapses.
- Change the letter of the system hard drive
- Turn on "high contrast" and other accessibility options.
- Change localization settings

Multivariable relationships

Any relationship between two variables is an opportunity for a relationship failure:

Examples of test ideas:

- Test with values that are individually valid but invalid together (e.g. February 30).
- Try similar things with dissimilar objects together (e.g. copy, resize or move) graphics and text together

QUICKTESTS HAVE LIMITS

Some people (incorrectly) characterize exploratory testing as if it were primarily a collection of quicktests.

As test design tools, these are like good candy:

- Yummy,
- Popular,
- Impressive, but
- Not very nutritious. (They don't take you to the deeper issues of the program.)

Quicktests are a great way to <u>start</u> testing a product.

Summary: Quicktests & risk-based testing

For the test designer, the essence of risk-based testing is:

a. Imagine how the product can fail
b. Design tests to expose these (potential) failures.

We've seen how quicktests address these tasks:

a. Use your experience (or the experience of others) to build a list of failures that are commonplace across many types of programs
b. Design straightforward tests that are focused on these specific bugs.

Different approaches to risk

1. Quicktests
2. Exploratory Guidewords
3. Failure Mode & Effects Analysis
4. Project-level risks

GUIDEWORDS

Guidewords are widely used in HAZOPs (hazard & operability studies).

Typically, a team analyzes a system together, covering each part of the system under evaluation using the guide words as a list of risk ideas.

NSW GOVERNMENT | Planning

Hazardous Industry Planning Advisory
Paper No 8

HAZOP Guidelines

January 2011

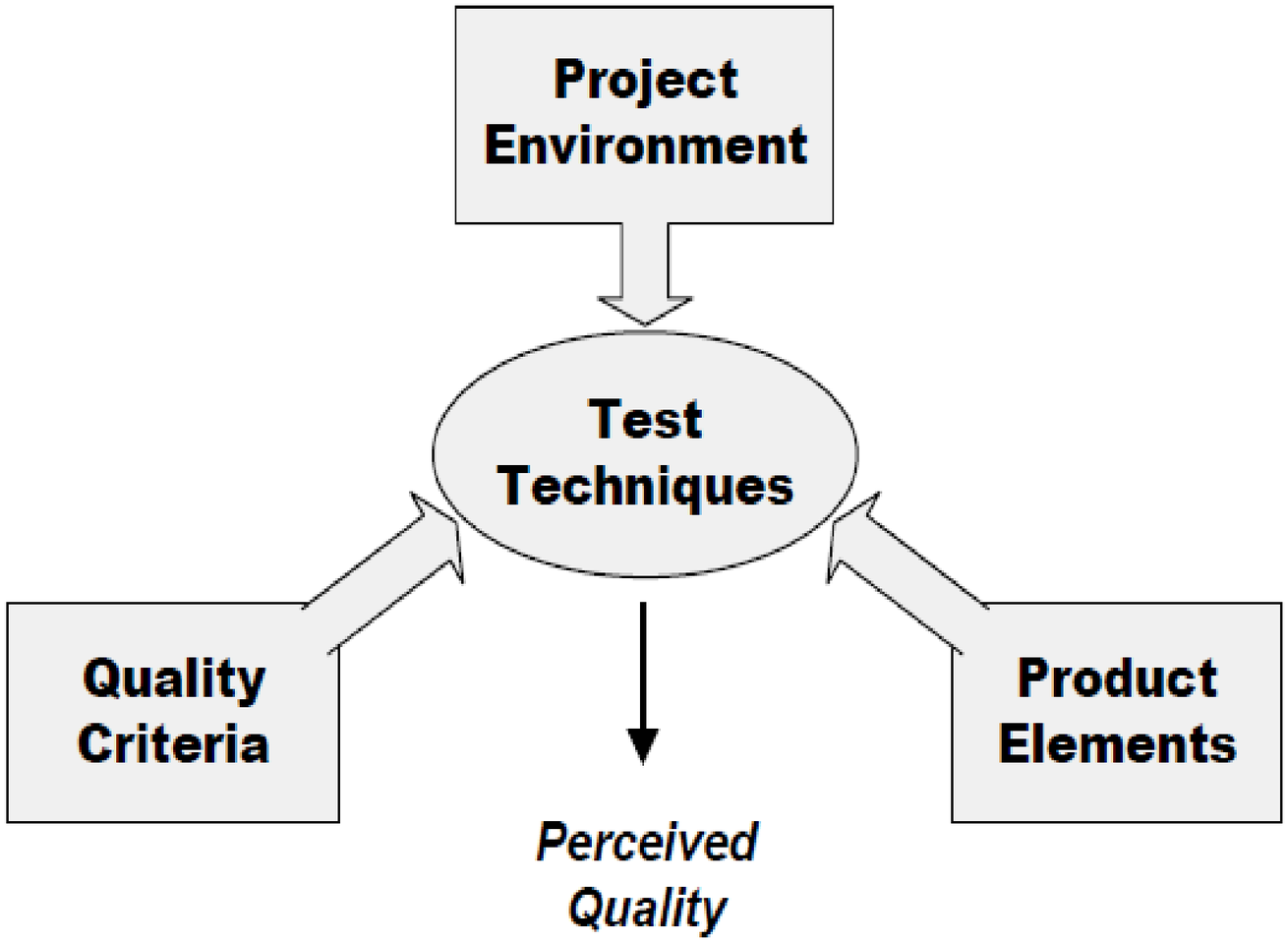
http://www.planning.nsw.gov.au/LinkClick.aspx?fileticket=HT4S8cZpZ3Y%3d&tabid=168&language=en-AU

See the papers in the Guidewords section of the reference list.

Designed by James Bach
james@satisfice.com
www.satisfice.com

Version 4.8
3/28/2006

Heuristic Test Strategy Model



http://www.satisfice.com/tools/satisfice-tsm-4p.pdf

Today's presentation of HTSM is just an overview. We come back to it in more detail in Lecture 3, and you will learn to apply it in your assignment.

Heuristic test strategy model

HTSM provides a customizable three-level collection of guide words.

Example:

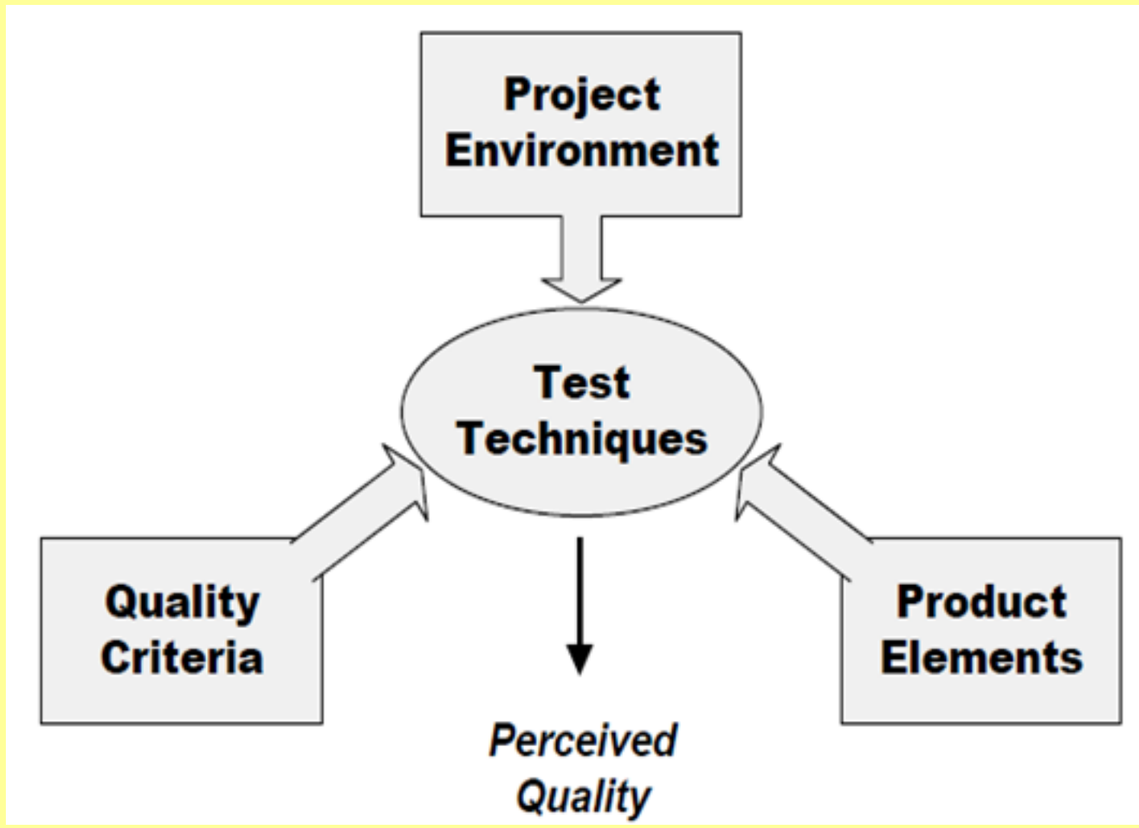
- Product elements
  - Structure
    - Interfaces

As with the HAZOPS use of guide words, the goal is to evaluate each part of the system under test from several directions, identifying a diverse collection of risks.

HTSM: PROJECT ENVIRONMENT

These categories lay out the context of the product, including factors that constrain what can be done in testing or that facilitate testing or test management.

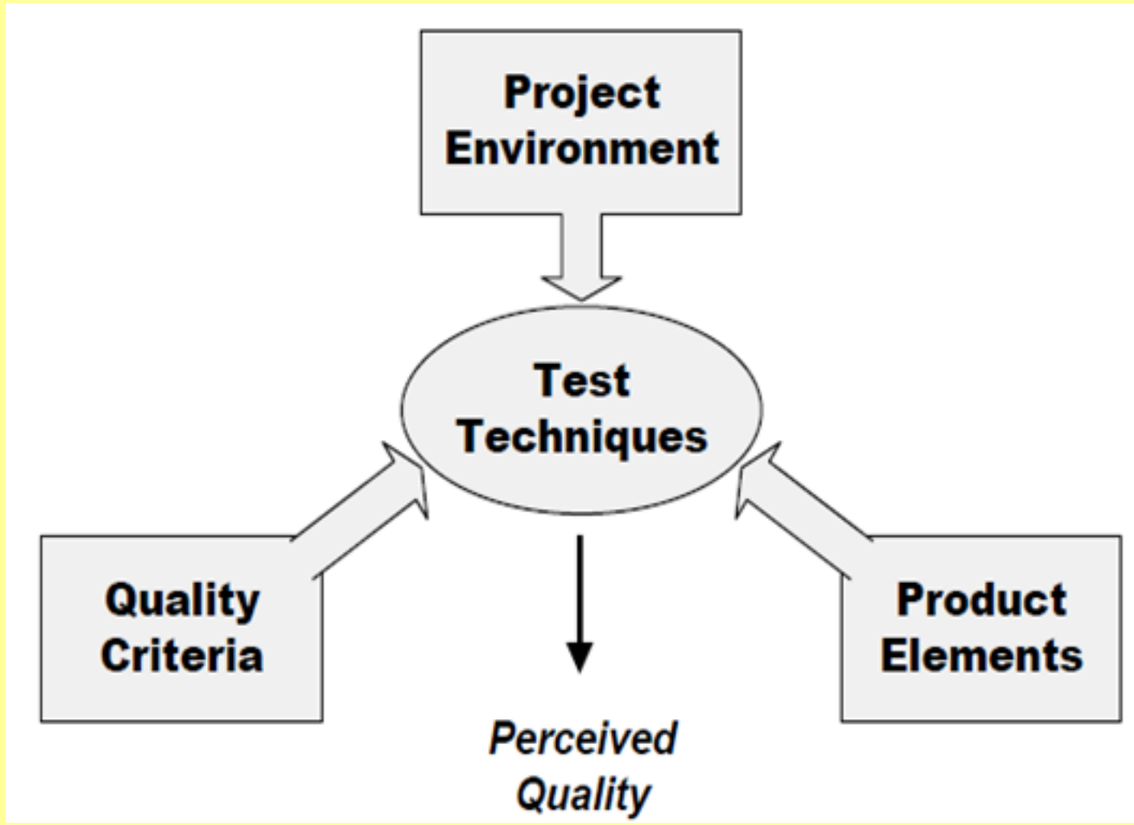
- Customers
- Information
- Developer relations
- Test team
- Equipment & tools
- Schedule
- Test items
- Deliverables



HTSM: PRODUCT ELEMENTS

These categories lay out the content of the application under test. This is what you're testing.

- Structure
- Functions
- Data
- Platform
- Operations
- Time



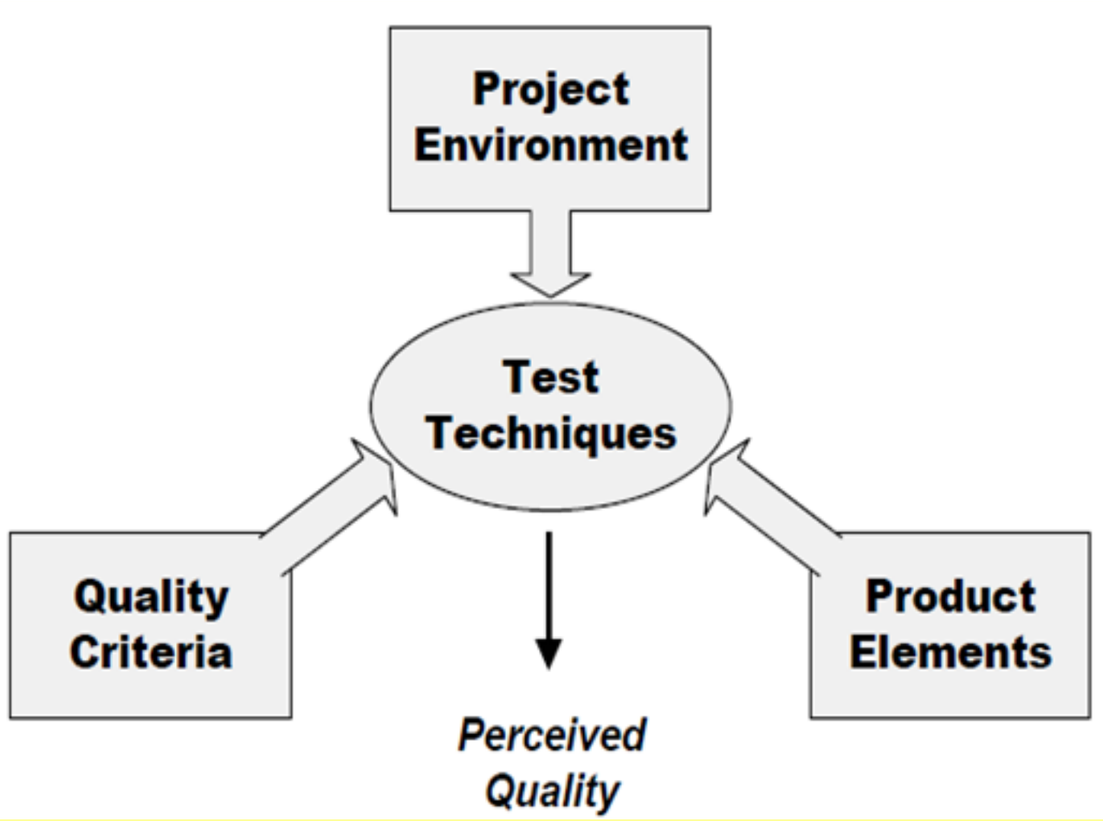
HTSM: Quality criteria

Operational criteria

- Capability
- Reliability
- Usability
- Security
- Scalability
- Performance
- Installability
- Compatibility

Development criteria

- Supportability
- Testability
- Localizability
- Maintainability
- Portability



USING HTSM TO GUIDE TESTING

- Pick a guide word (e.g. *interfaces*).
- Identify "all" aspects of the program that match the guide word
- One by one, what could go wrong with each?
- You can also combine guide words
  - From different categories (for example: *Product elements: interfaces* with *Project environment: customers*.)
  - From the same category (for example: *Product elements: interfaces* with *Product elements: data*.)

This is a useful structure for exploratory testing.

Different approaches to risk

1. Quicktests
2. Exploratory Guidewords
3. Failure Mode & Effects Analysis
4. Project-level risks

For the test designer, the essence of risk-based testing is:

a) Imagine how the product can fail

b) Design tests to expose these (potential) failures.

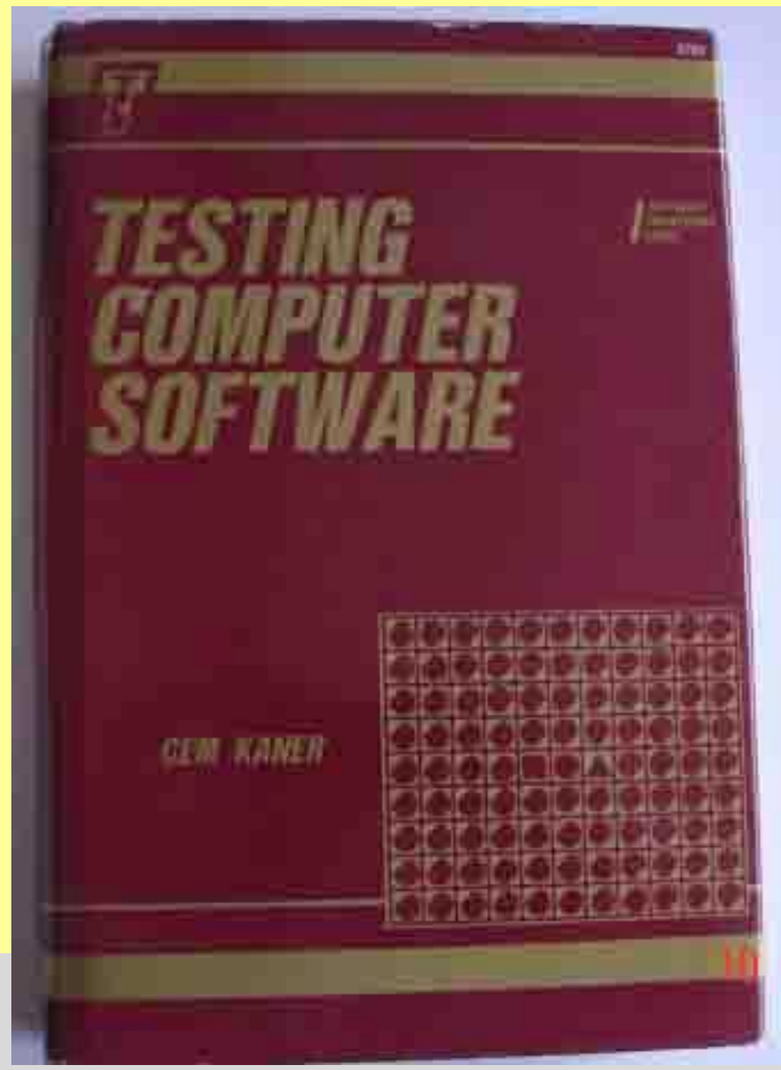
Failure mode lists / Risk catalogs / Bug taxonomies

- A failure mode is, essentially, a way that the program could fail.
- One way to structure risk-based testing is with a list of failure modes
  - also called a risk catalog
  - or a bug taxonomy
- Use each failure mode as a test idea (something to create a test for)

Our first list of quicktests was derived from a bug catalog

Testing Computer Software listed almost 500 common bugs. We used the list for:

- Generating test ideas (Getting unstuck)
- Structuring exploratory testing
- Auditing test plans
- Training new staff into risk-oriented thinking



A revised (2nd ed) version of Kaner (1988)'s list is available:

http://logigear.com/articles-by-logigear-staff/445-common-software-errors.html

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

From Bach (1999) Heuristic Risk-Based Testing.

Building a failure mode catalog

Giri Vijayaraghavan and Ajay Jha followed similar approaches in developing their catalogs:

- Used HTSM as a starting structure
- Filled-in real life examples of failures from magazines, web discussions, some corporations' bug databases, interviews with people who had tested their class of products.
- Extrapolated to other potential failures
- Extended to potential failures involving interactions among components

- A Risk Catalog for Mobile Applications, http://www.testingeducation.org/articles/AjayJha_Thesis.pdf
- A Taxonomy of E-Commerce Risks and Failures, http://www.testingeducation.org/a/tecrf.pdf
- Bug taxonomies: Use them to generate better tests. http://www.testingeducation.org/a/bugtax.pdf

Failure mode & effects analysis

FMEA is a more common, formalized approach to risk-based evaluation of many types of products.

Failure modes

Consider the product in terms of its components. For each component

- Imagine how it could fail (failure modes). For 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?

Failure mode analysis is an effective vehicle for generating test idea lists.

Failure mode & effects analysis

Widely used for safety analysis.

Effect analysis

- For each failure mode:
  - Who would that failure impact?
  - 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

"Accident investigators tend to take an expansive approach when determining the "cause" of an accident. Aware that regulations are influenced by accident reports, investigators often seek to effect the greatest possible change. "It's better if you don't find the exact cause because then only one thing will get fixed," according to an NTSB investigator. Instead, for every serious accident the NTSB recommends a laundry list of changes in FAA regulations." Cheit (1990, 71).

USING FAILURE MODE CATALOGS

Generate test ideas

- Find a potential 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.

USING FAILURE MODE CATALOGS

Provide a structure for exploratory testing

Exploratory software testing is

- a style of software testing that
- emphasizes the personal freedom and responsibility of the individual tester
- to continually optimize the value of her work
- by treating
  - test-related learning,
  - test design,
  - test execution, and
  - test result interpretation as
- mutually supportive activities that run in parallel throughout the project.

As you learn more about how the product can fail, design new tests to explore potential failures.

And do new research (or follow new hunches) to find more new categories of ways the product can fail.

USING FAILURE MODE CATALOGS

Audit test plans

The "test plan" is a document that describes the planned testing. Often, this document is very detailed. In some companies use the test plan to fully specify the testing that will be done.

- Pick categories to sample from the test idea list
- 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.

USING FAILURE MODE CATALOGS

Training new staff into risk-oriented thinking

- Expose staff to what can go wrong
- Challenge them to design tests that could trigger those failures

Different approaches to risk

1. Quicktests
2. Exploratory Guidewords
3. Failure Mode & Effects Analysis
4. Project-level risks

For the test designer, the essence of risk-based testing is:

a) Imagine how the product can fail

b) Design tests to expose these (potential) failures.

Project-level risk analysis

Project-level risk analyses consider what might make the project as a whole fail.

Project risk management involves

- Identifying issues that might cause the project to fail or fall behind schedule or cost too much or alienate key stakeholders
- Analyzing potential costs associated with each risk
- Developing 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)

CLASSIC, PROJECT-LEVEL RISK ANALYSIS

Acrobat Reader - [tutorial from sei 2.pdf]

File Edit View Tools Window Help

Categories of Risk Sources

- Mission and goals
- Decision drivers
- Organization management
- Customer / end user
- Budget / cost
- Schedule
- Project characteristics
- Development process
- Development environment
- Personnel
- Operational environment
- New technology

Project Consequences

- Cost overruns
- Schedule slips
- Inadequate functionality
- Canceled projects
- Sudden personnel changes
- Customer dissatisfaction
- Loss of company image
- Demoralized staff
- Poor product performance
- Legal proceedings

TeraQuest

SEPG Risk Workshop
© 1998 TeraQuest

Page 12 of 53 145% 9.54 x 7.28 in

The problem for our purposes is that the traditional analysis at this level is more oriented to project managers. It doesn't give you much guidance as to how or what to test.

Useful material available at

https://seir.sei.cmu.edu/seir

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.

As testers, you can use risks associated with the running of the project to suggest specific ideas that can guide your testing.

Project risk heuristics: Where to look for errors

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.

Fatigue: tired people make mistakes.

Distributed team: a far flung team often communicates less or less well.

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

Weinberg (1993, Ch. 10) provides useful insights into the challenges of rushed work and late changes.

Project risk heuristics: Where to look for errors

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.

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

Project risk heuristics: Where to look for errors

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.

Project risk heuristics: Where to look for errors

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

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.

Project risk heuristics: Where to look for errors

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.

Little system testing: untested software will fail.

Little unit testing: programmers normally find and fix most of their own bugs.

Project risk heuristics: Where to look for errors

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.

Project risk heuristics: Where to look for errors

Untestable: anything that requires slow, difficult or inefficient testing is probably undertested.

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.

Project risk heuristics: Where to look for errors

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.

Review

- Test design:
  - test cases & techniques
  - testing strategy: based on …
    - Information objectives
    - context factors
- Risk-based testing
  - Quicktests
  - Exploratory guidewords and the HTSM
  - Failure mode & effects analysis
  - Project-level risks

End of Lecture 2

Black Box Software Testing: Introduction to Test Design: Lecture 3: Spec-Based Testing

Cem Kaner, J.D., Ph.D.
Professor of Software Engineering: Florida Tech

Rebecca L. Fiedler, M.B.A., Ph.D.
President: Kaner, Fiedler & Associates

These notes are partially based on research that was supported by NSF Grants EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers" and CCLI-0717613 "Adaptation & Implementation of an Activity-Based Online or Hybrid Course in Software Testing." 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.

OVERVIEW OF THE COURSE

	Technique	Context / Evaluation
1	Function testing & tours.	A taxonomy of test techniques.
2	Risk-based testing, failure mode analysis and quicktests	Testing strategy. Introducing the Heuristic Test Strategy Model.
3	Specification-based testing.	(… work on your assignment …)
4	Use cases and scenarios.	Comparatively evaluating techniques.
5	Domain testing: traditional and risk-based	When you enter data, any part of the program that uses that data is a risk. Are you designing for that?
6	Testing combinations of independent and interacting variables.	Combinatorial, scenario-based, risk-based and logical-implication analyses of multiple variables.

LECTURE 3 READINGS

Required reading

- Bach, J. (2006). Heuristic test strategy model, Version 4.8. http://www.satisfice.com/tools/satisfice-tsm-4p.pdf

Recommended reading

- Adler, M., & Van Doren, C. (1972). How to Read a Book. Touchstone
- Gause, D.C., & Weinberg, G.M. (1989). Exploring Requirements: Quality Before Design. Dorset House
- Moon, B.M., Hoffman, R.R., Novak, J.D., & Canas, A.J. (Eds., 2011). Applied Concept Mapping: Capturing, Analyzing, and Organizing Knowledge. CRC Press

Your class might use other readings, but we had these in mind when creating this lecture.

WHAT IS SPEC-BASED TESTING?

1. Activities focused on testing the product against claims made in specifications.

   This is what we mean by spec-based testing.

2. Testing focused on logical relationships among variables that are often detailed in specifications.

   We study this as multivariable testing.

3. Activities focused on proving that statements in a specification (and code that implements the statements) are logically correct.

   This is taught in more theoretical courses.

CRITICAL QUESTIONS

1. What is the specification?
2. Why did they create it?
3. Who are the stakeholders?
4. What are you trying to learn or achieve with the spec?
5. What are the consequences of nonconformity?
6. What claims does the specification make?
7. What ambiguities must be resolved?
8. How should you use it to create tests?

What is the specification?

- Include **any** document that:
  - describes the product, and
  - drives development, sale, support, use, or purchase of the product, and
  - either
    - was created by the maker or other vendor of the product OR
    - would be accepted by the maker or other vendor of the product as an accurate or controlling description.

The complete set of specifications can include documents created by third parties or after the product is finished or by third parties.

WHAT IS THE SPECIFICATION?

What is the scope of this specification?

- Some specs cover the entire product, others describe only part of it (such as error handling).
- Some specs address the product from multiple points of view, others only from one point of view.

What is the specification?

Do you have the right specification?

- Do you have the current version?
- Is the spec kept under source control?
- How do you verify the version?

Is this a stable specification?

- Is the product under change control?
- Is the spec under change control?
  - Should it be?

Implicit specifications

Some aspects of the product are clearly understood, but not described in detail in the formal specifications because:

- they are determined by controlling cultural or technical norms (and often described in documents completely independent of this product), or
- they are defined among the staff, perhaps in some other document.

Finding documents that describe these implicit specifications is useful: Rather than making an unsupported statement like “this is inappropriate” or “users won’t like it”, you can use implicit specifications to justify your assertions.

Examples of implicit specifications

- Published style guide and UI standards
- Published standards (such as C-language or IEEE Floating Point)
- 3rd party product compatibility test suites
- Localization guide (probably published for localizing products on your platform.)
- Published regulations

EXAMPLES OF IMPLICIT SPECIFICATIONS

- Marketing presentations (e.g. documents that sell the concept of the product to management)
- Internal memos (e.g. project mgr. to engineers, describing feature definitions)
- User manual draft (and previous version's manual)
- Product literature (advertisements and other promotional documents)
- Sales presentations
- Software change memos that come with each new internal version of the program

Look for in-house documents that describe the product to influential stakeholders.

("In-house" means created by the company for its own use.)

Examples of implicit specifications

- Bug reports (responses to them)
- Look at customer call records from the previous version. What bugs were found in the field?
- Usability test results (and corporate responses to them)
- Beta test results (and corporate responses to them)
- 3rd party tech support databases, magazines and web sites with:
  - discussions of bugs in your product
  - common bugs in your niche or on your platform
  - discussions of how some features are supposed (by some) to work.

Examples of implicit specifications

- Reverse engineer the program
- Look at header files, source code, database table definitions
- Prototypes, and lab notes on the prototypes
- Interview people, such as
  - development lead
  - tech writer
  - customer service
  - subject matter experts
  - project manager
  - development staff from the last version.

Examples of implicit specifications

- Specs and bug lists for all 3rd party tools that you use
  - Example: If your company develops software for the Windows platform, the Microsoft Developer Network has lots of relevant info
- Get lists of compatible equipment and environments
  - Interface specifications
  - Protocol specifications
- Reference materials that can be used as oracles for the content that comes with the program (e.g. use an atlas to check your on-line geography program)

Examples of implicit specifications

- Look at competing products:
  - Similarities and differences between the benefits and features offered by the products
  - How the other products describe their design, capabilities and behaviors
  - What weaknesses they have, what bugs they have or publicly fixed
- Make precise comparisons with products you emulate. If product X is supposed to work "just like" Y, compare X and Y thoroughly.

Anything that drives people's expectations of the product is a (explicit or implicit) specification.

CRITICAL QUESTIONS

1. What is the specification?
2. Why did they create it?
3. Who are the stakeholders?
4. What are you trying to learn or achieve with the spec?
5. What are the consequences of nonconformity?
6. What claims does the specification make?
7. What ambiguities must be resolved?
8. How should you use it to create tests?

Questions like these frame the context analysis behind specification-driven testing. There is no one best way to test against a specification. The details of your testing are determined by your context.

Why did they create it?

- Add an enforceable description to a contract for custom software?
- Present a product vision? (Details illustrate the intent of the product but will change in implementation.)
- Provide an authoritative description for development?
- Provide a description that marketers, sales or advertisers can rely on?
- Facilitate and record agreement among stakeholders? About specific issues or about the whole thing?
- Provide support material for testers / tech support staff / technical writers?
- Comply with regulations?

CRITICAL QUESTIONS

1. What is the specification?
2. Why did they create it?
3. Who are the stakeholders?
4. What are you trying to learn or achieve with the spec?
5. What are the consequences of nonconformity?
6. What claims does the specification make?
7. What ambiguities must be resolved?
8. How should you use it to create tests?

Who are the stakeholders?

- Who is the champion of this document?
- Who cares whether the program matches the spec, and why do they care?
- Who cares if the spec is kept up to date and correct?
- Who doesn't care if it is kept up to date?
- Who is accountable for its accuracy and maintenance?
- Who will have to deal with corporate consequences if it is inaccurate?
- Who will invest in your developing an ability to understand the specification?

CRITICAL QUESTIONS

1. What is the specification?
2. Why did they create it?
3. Who are the stakeholders?
4. What are you trying to learn or achieve with the spec?
5. What are the consequences of nonconformity?
6. What claims does the specification make?
7. What ambiguities must be resolved?
8. How should you use it to create tests?

What are you trying to learn or achieve with the spec?

- Learn about the product?
- Support project manager's use of the spec as a driver of the project?
- Prevent problems (via design review) before they are coded in?
- During test planning, identify testing issues before you get code?
- Source of test ideas while testing?
- Source of evidence that product behaviors are or are not bugs?
- Manage contract-related risks?
- Manage regulatory risks?
- Help company assess product drift?

CRITICAL QUESTIONS

1. What is the specification?
2. Why did they create it?
3. Who are the stakeholders?
4. What are you trying to learn or achieve with the spec?
5. What are the consequences of nonconformity?
6. What claims does the specification make?
7. What ambiguities must be resolved?
8. How should you use it to create tests?

Consequences of nonconformity?

Nonconformity with the specification will sometimes carry legal implications:

- In custom software, a spec that is part of the contract creates a warranty. The non-conforming product is defective and the buyer can refuse to pay or demand a discount.
- In software sold to the public, specs create warranties (whether the vendor intends them as warranties or not): **http://www.kaner.com/pdfs/liability_sigdoc.pdf**
- A claim that a product is compatible with another creates warranties that the product won't fail compatibility tests: **http://www.kaner.com/pdfs/liability.pdf**

CRITICAL QUESTIONS

1. What is the specification?
2. Why did they create it?
3. Who are the stakeholders?
4. What are you trying to learn or achieve with the spec?
5. What are the consequences of nonconformity?
6. What claims does the specification make?
7. What ambiguities must be resolved?
8. How should you use it to create tests?

What claims does the spec make?

- Specifications can run thousands of pages
  - spread across multiple documents
  - which incorporate several other documents by reference
  - using undefined, inconsistently defined or idiosyncratically defined vocabulary
- Specification readers often suffer severe information overload.

Active reading skills and strategies are essential for effective specification analysis

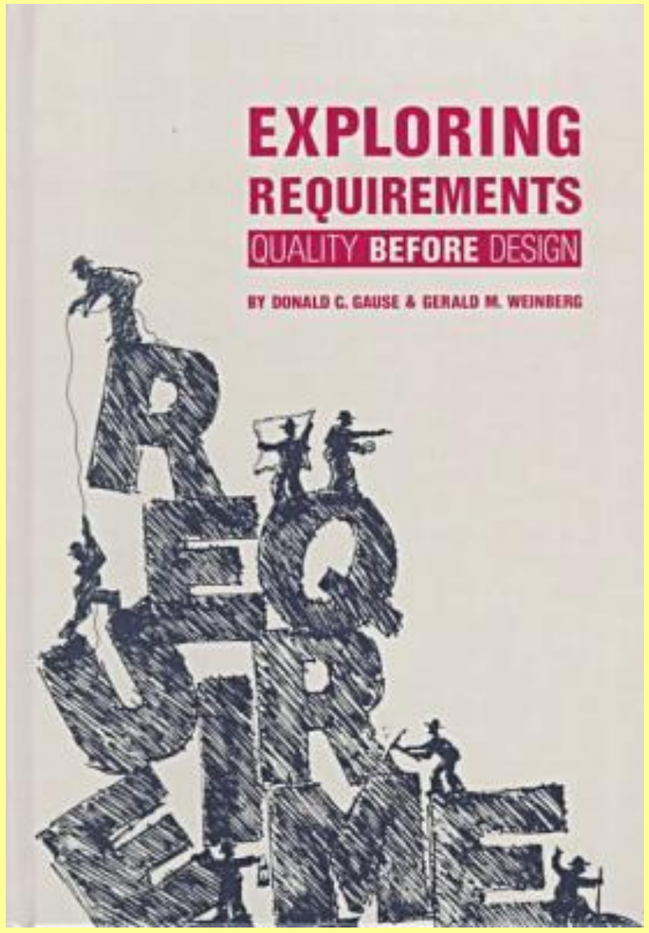
Active reading (Example)

Prioritize what you read, by

- Surveying (read table of contents, headings, abstracts)
- Skimming (read quickly, for overall sense of the material)
- Scanning (seek specific words or phrases)

Search for information in the material you read, by

- Asking information-gathering questions and search for their answers
- Creating categories for information and read to fill in the categories
- Questioning / challenging / probing what you're reading

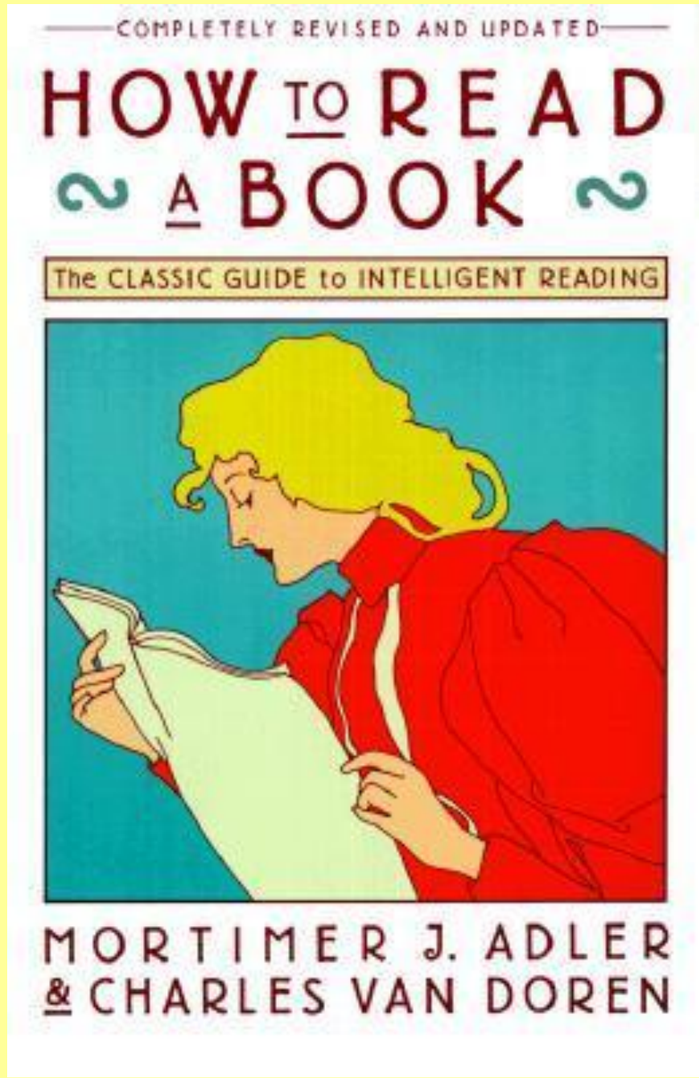


Gause / Weinberg is a superb source for context-free questions

Active reading (Example)

Organize it

- Read with a pen in your hand
- If you underline or highlight, don't do so until AFTER you've read the section
- Make notes as you go
  - Key points, Action items, Questions, Themes, Organizing principles
- Use concise codes in your notes (especially on the book or article). Make up 4 or 5 of your own codes. These two are common, general-purpose:
  - ? means I have a question about this
  - ! means new or interesting idea



Active reading (Example)

Organize it

- Spot patterns and make connections
  - Create information maps
- Relate new knowledge to old knowledge

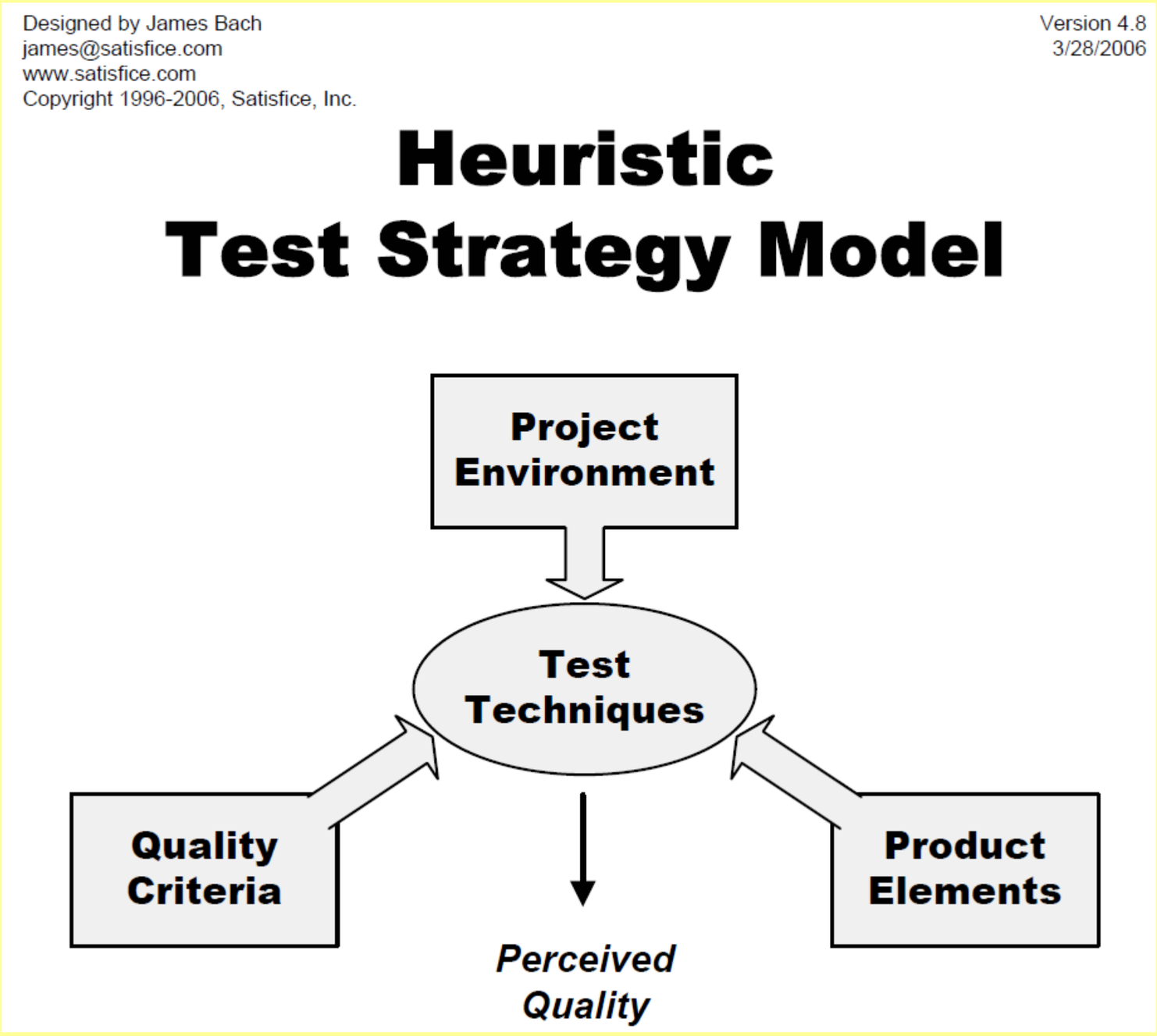
Explain it

- The core ideas, the patterns, the relationships…
- To yourself or to someone else

Plan for your retention of the material

- Cubing as a post-reading exercise www.douglasesd.k12.or.us/LinkClick.aspx?fileticket=6FrvrocjBnk%3d&tabid=1573
- SQ3R (survey / question / read / recite / review)
- Archival notes

Using the Heuristic Test Strategy Model for active reading



http://www.satisfice.com/tools /satisfice-tsm-4p.pdf

USING HTSM FOR ACTIVE READING

This model provides a structure for:

- Sorting / classifying a complex body of information
- Generating test ideas
  - about the classified information or about combinations of the classified information
  - Guide words (HAZOPS)
    - We talked about this when we covered risk-based testing
  - Generative taxonomy

Using Bach's HTSM for active reading

- Every statement in the specification describes some aspect(s) of the project or product
  - **Product Elements:** things you can test.
  - **Project Factors:** aspects of the project that facilitate or constrain the testing effort.
  - **Quality Criteria:** what stakeholders value about the product. Quality criteria are multidimensional, and often incompatible with each other. A specific criterion might be essential for one product and not very important for another.

Create a map of the HTSM, then sort every statement of interest into the structure created by the map.

Concept maps

- Mind Manager:
  http://www.mindjet.com
- NovaMind:
  http://www.novamind.com
- Inspiration
  http://www.inspiration.com
- XMind
  http://www.xmind.net

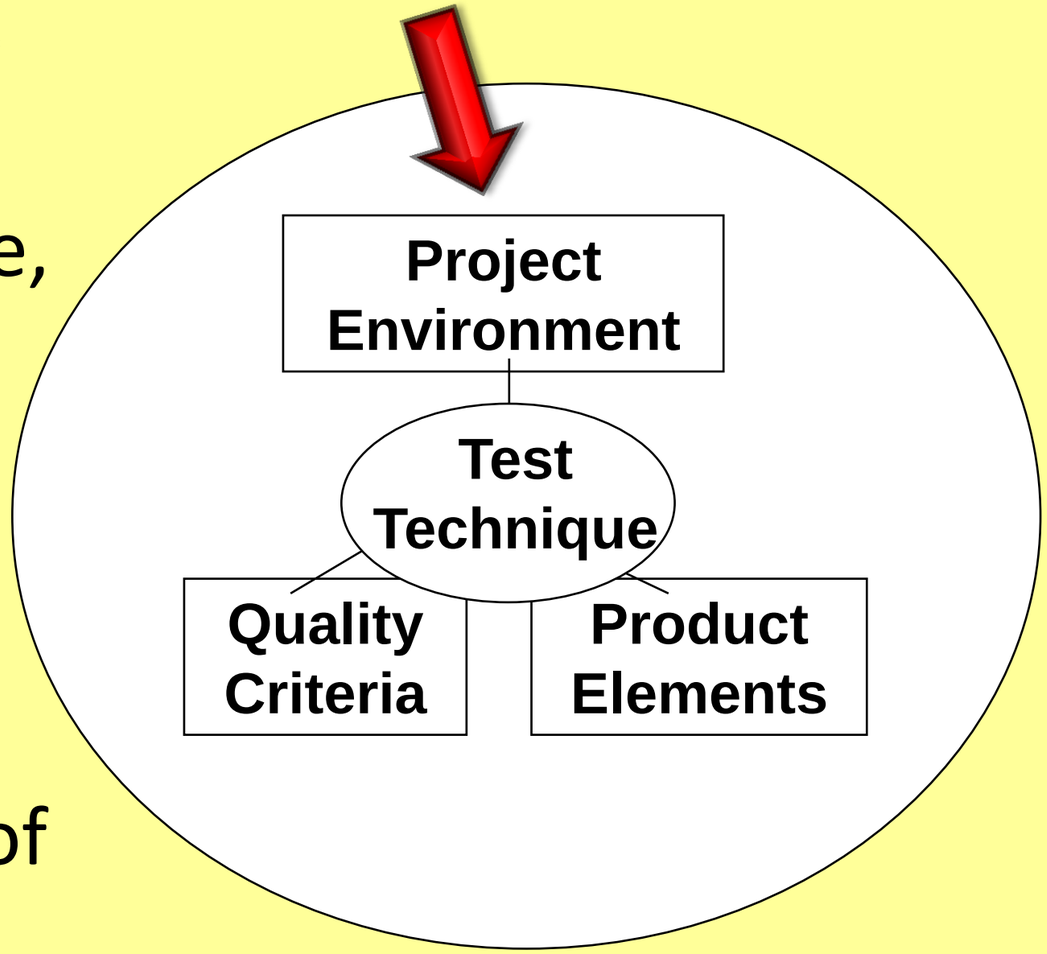
For a very useful list of tools, see Wikipedia:

http://en.wikipedia.org/wiki/Concept_mapping_program

CREATE A MAP OF THIS MODEL

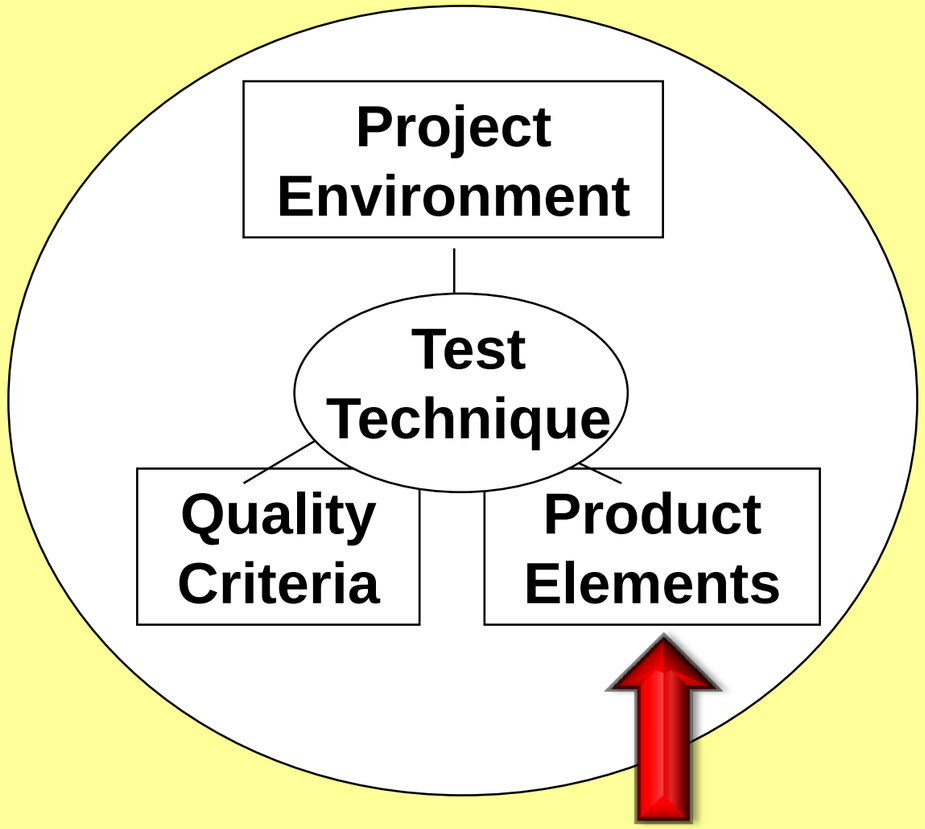
The Map: Project Environment

- **Customers**: Any client of the test project.
- **Information**: Information about the product or project is needed for testing.
- **Developer Relations**: How you get along with the programmers.
- **Test Team**: Anyone who will perform or support testing
- **Equipment & Tools**: Hardware, software, or documents required to administer testing.
- **Schedule**: Sequence, duration, and synchronization of project events.
- **Test Items**: The product to be tested.
- **Deliverables**: The observable products of the test project.



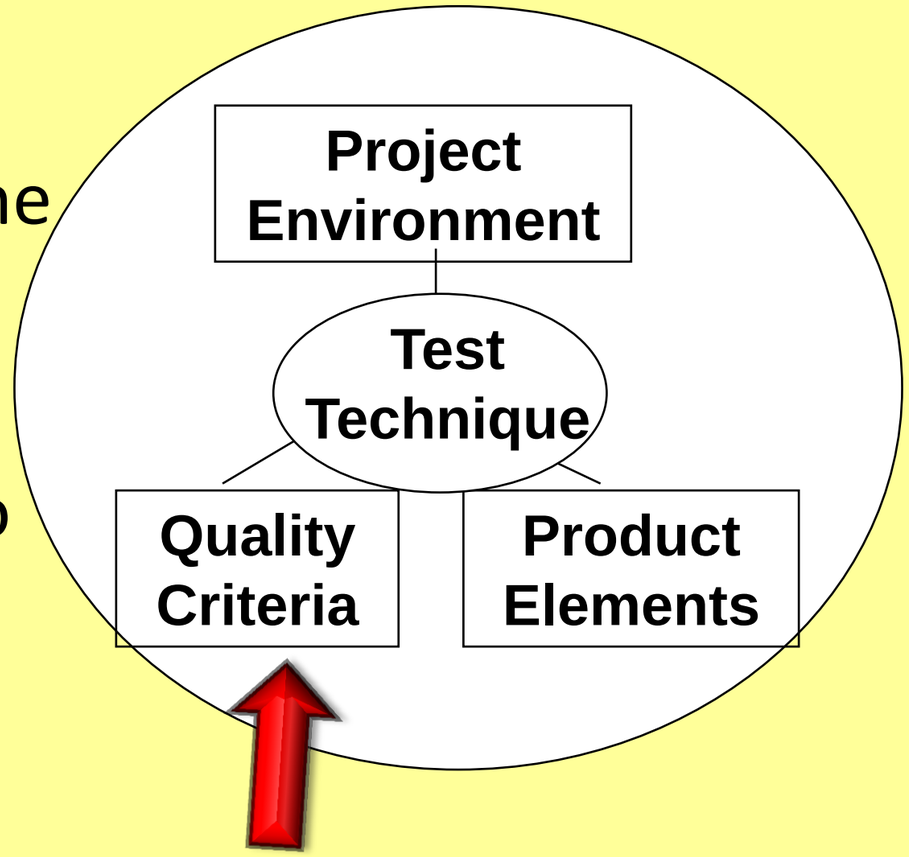
The Map: Product Elements

- **Structure**: Everything that comprises the physical product
- **Functions**: Everything the product does
- **Data**: Everything the product processes
- **Platform**: Everything on which the product depends (and that is outside your project)
- **Operations**: How the product will be used
- **Time**: Any relationship between the product and time



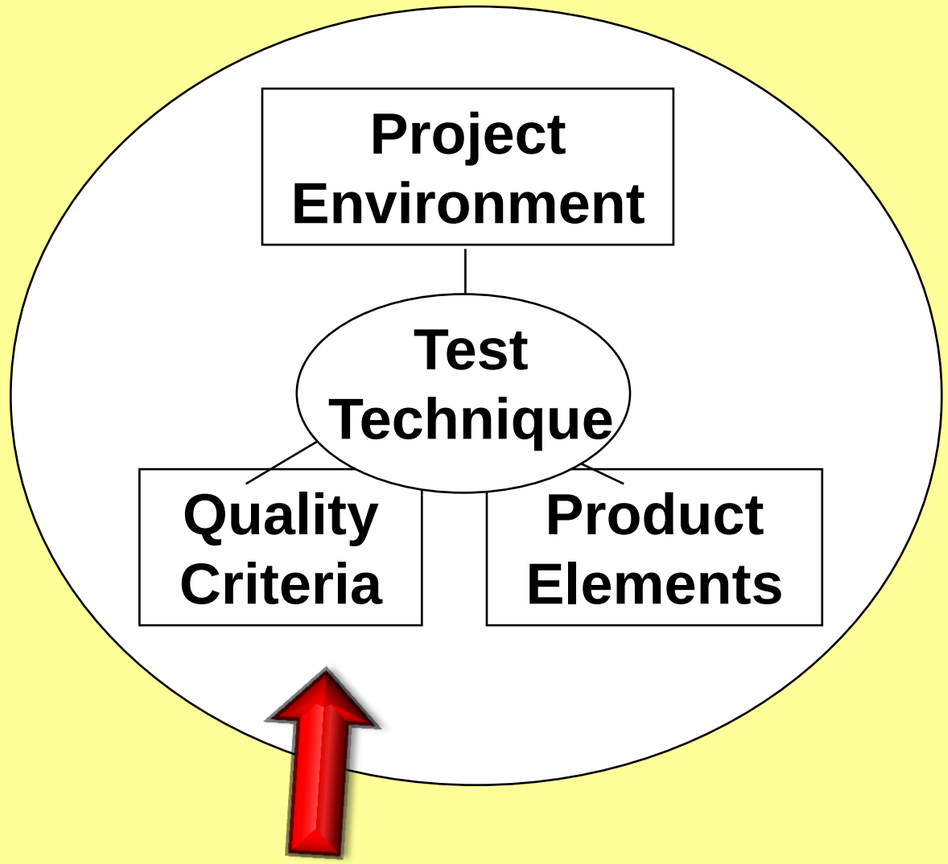
The map: Quality Criteria: Operational Criteria

- **Capability**: Can it perform the required functions?
- **Reliability:** Will it work well and resist failure in all required situations?
- **Usability:** How easy is it for a real user to use the product?
- **Security:** How well is the product protected against unauthorized use or intrusion?
- **Scalability:** How well does the deployment of the product scale up or down?
- **Performance:** How speedy and responsive is it?
- **Installability:** How easily can it be installed onto its target platforms?
- **Compatibility:** How well does it work with external components & configurations?



The map: Quality Criteria: 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 adapt the product for other places?



THE FULL MODEL HAS DEPTH

Project Environment

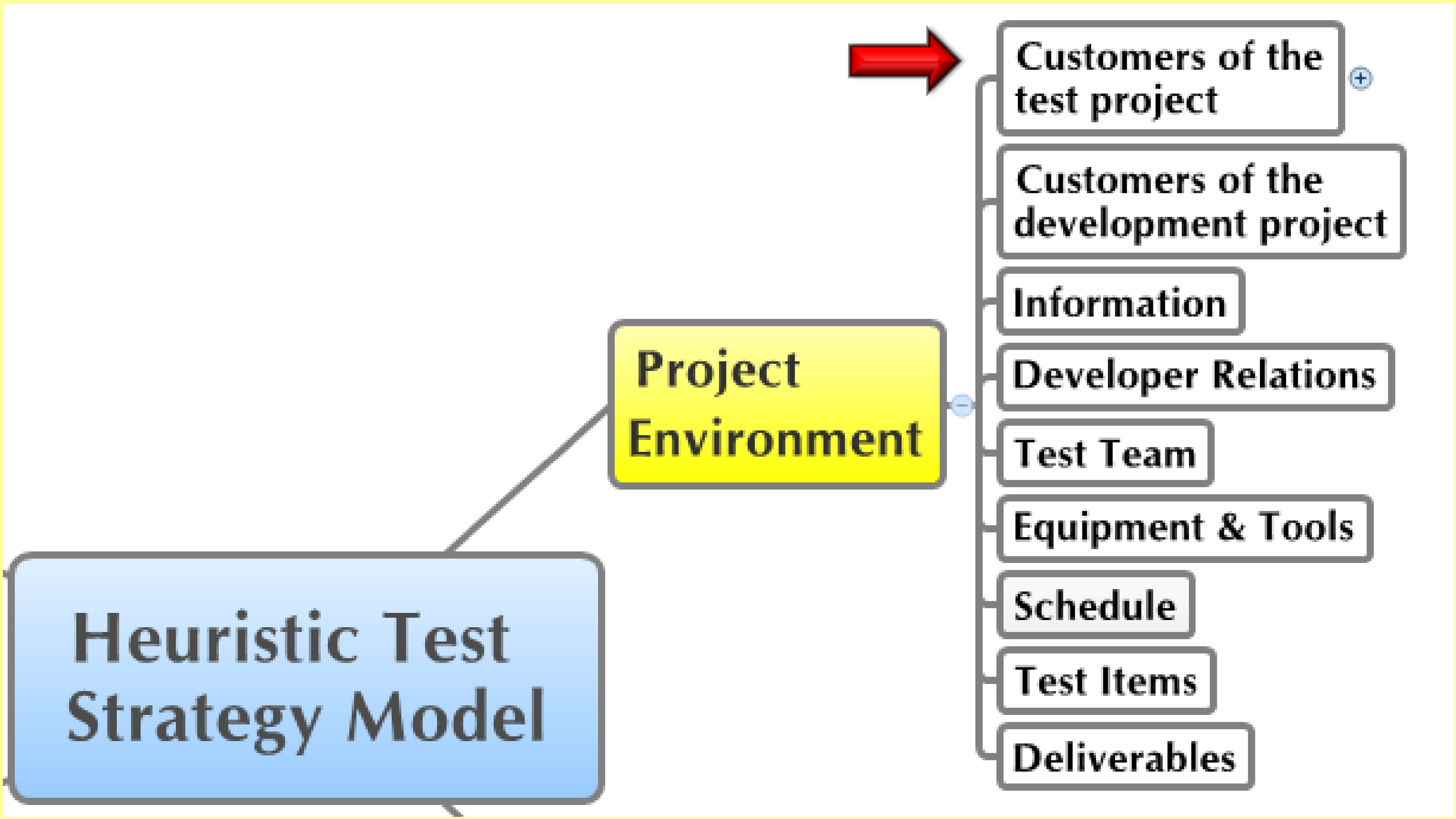
Creating and executing tests is the heart of the test project. However, there are many factors in the project environment that are critical to your decision about what particular tests to create. In each category, below, consider how that factor may help or hinder your test design process. Try to exploit every resource.

- **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 they can help you test.
  - 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.

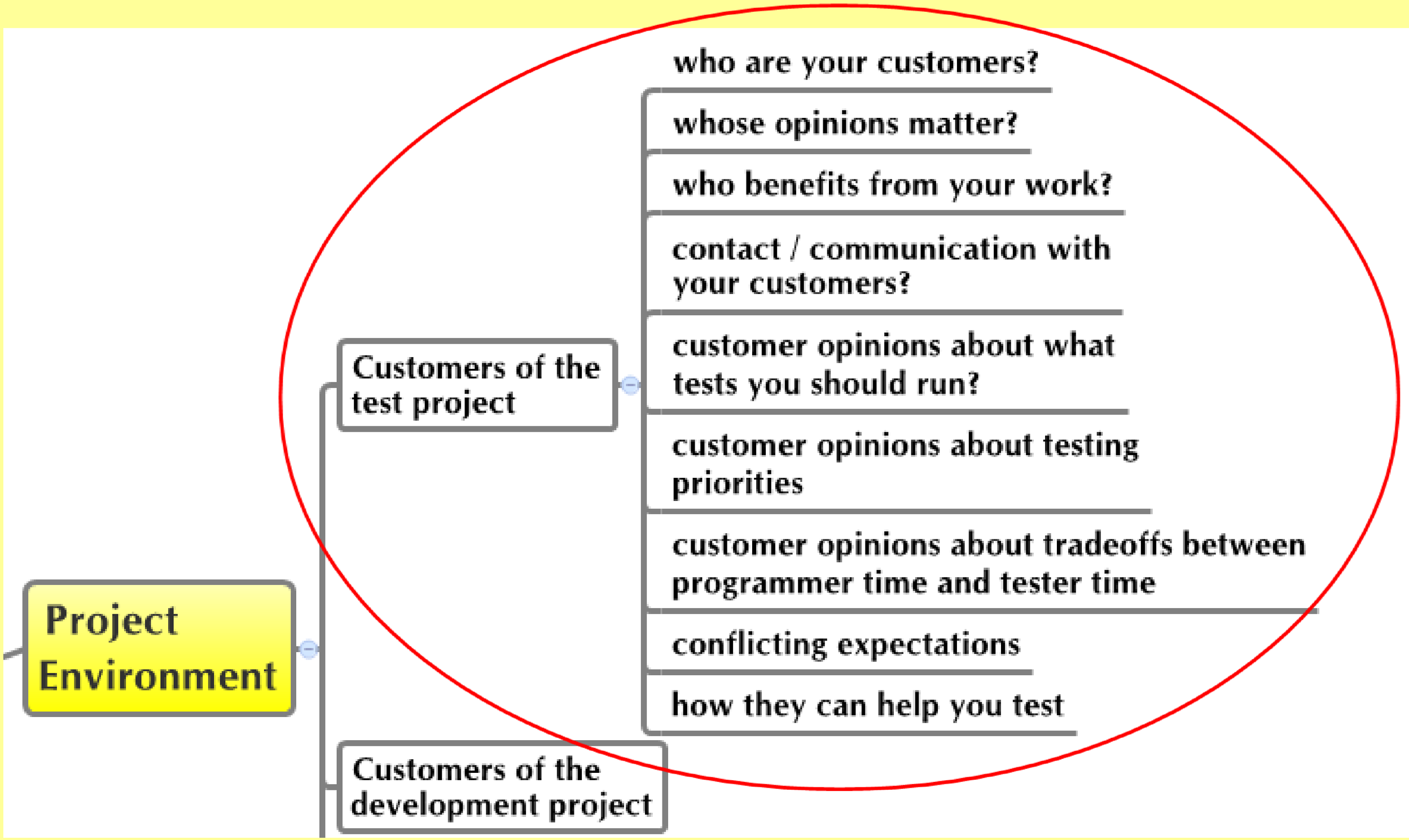
- **Information.** *Information about the product or project that is needed for testing.*
  - Are there any engineering documents available? User manuals? Web-based materials?
  - Does this product have a history? Old problems that were fixed or deferred? Pattern of customer complaints?
  - Do you need 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?
  - Is there any complex or challenging part of the product about which there seems strangely little information?

http://www.satisfice.com/tools/satisfice-tsm-4p.pdf

You can customize the model:



So add a level to the map



You can customize the model:

Customers: Any client of the test project.

Information: Information about the product or project is needed for testing.

Developer relations: How you get along with the programmers.

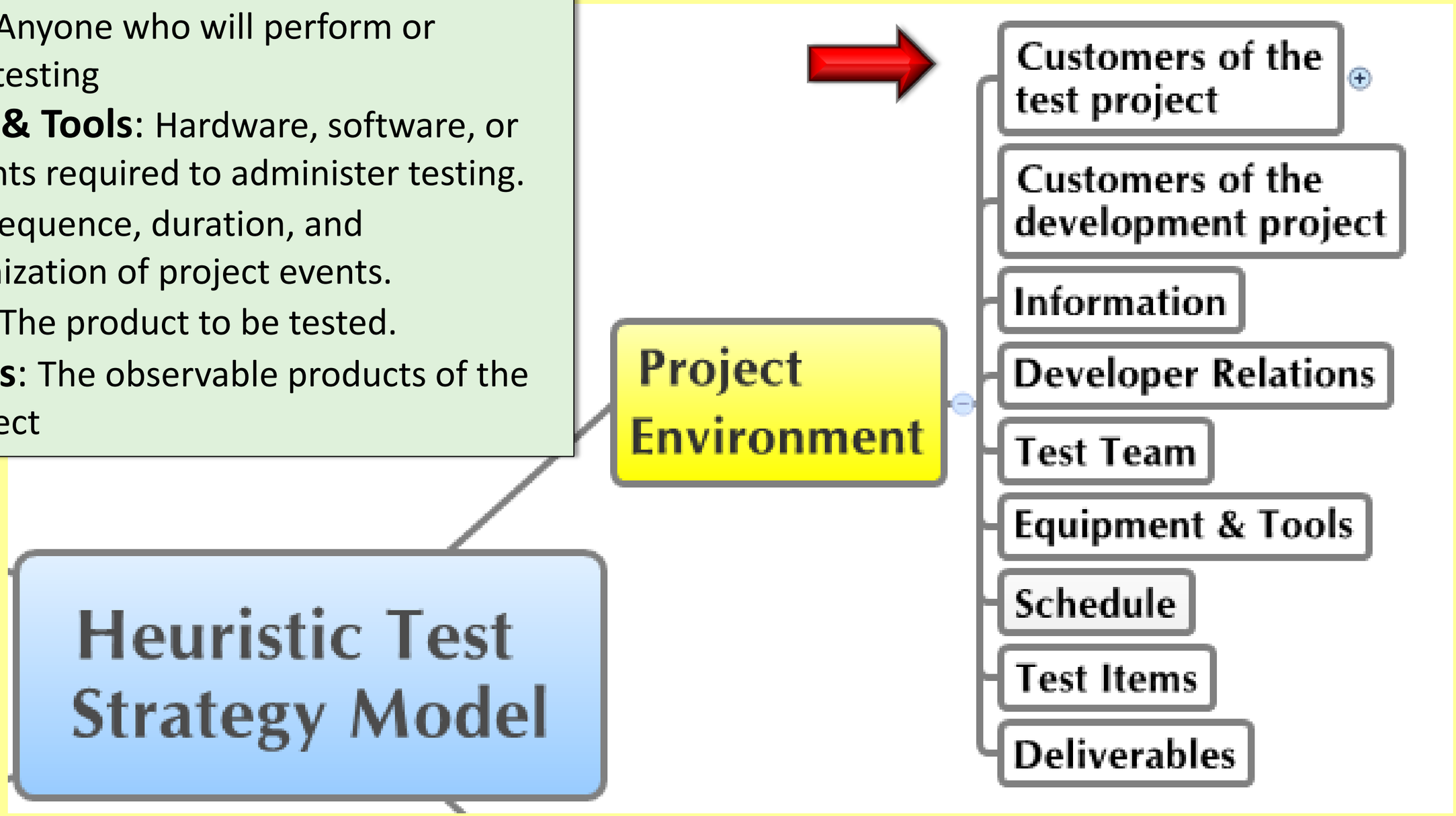
Test Team: Anyone who will perform or support testing

Equipment & Tools: Hardware, software, or documents required to administer testing.

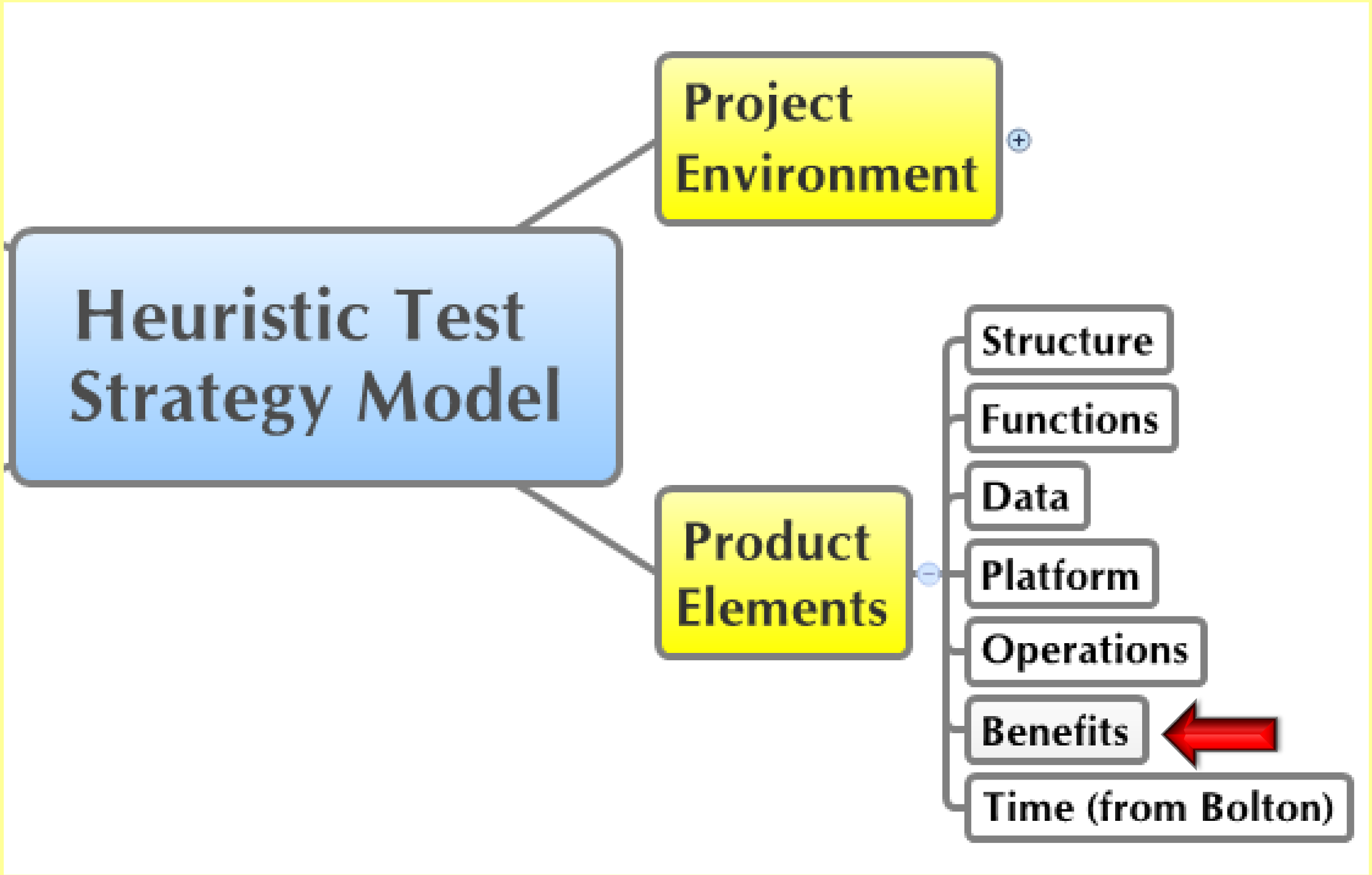
Schedule: Sequence, duration, and synchronization of project events.

Test Items: The product to be tested.

Deliverables: The observable products of the test project

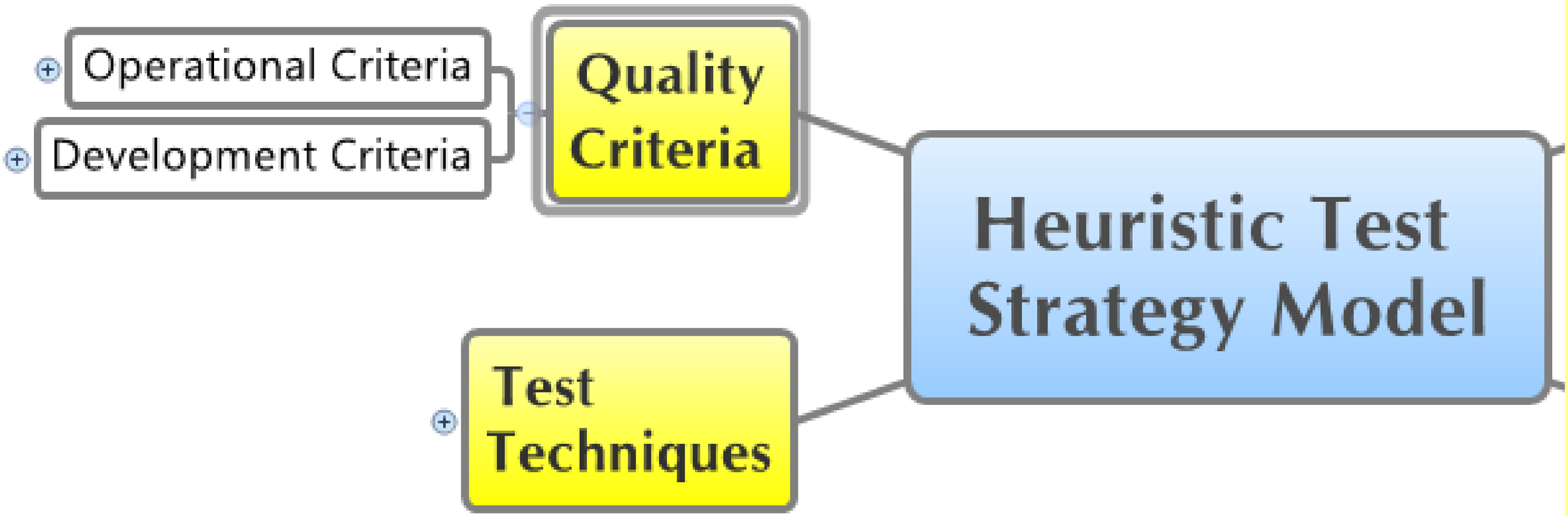


I added Benefits to Product Elements, because this helps me think about scenarios. I also added Mike Bolton's treatment of time and timing.



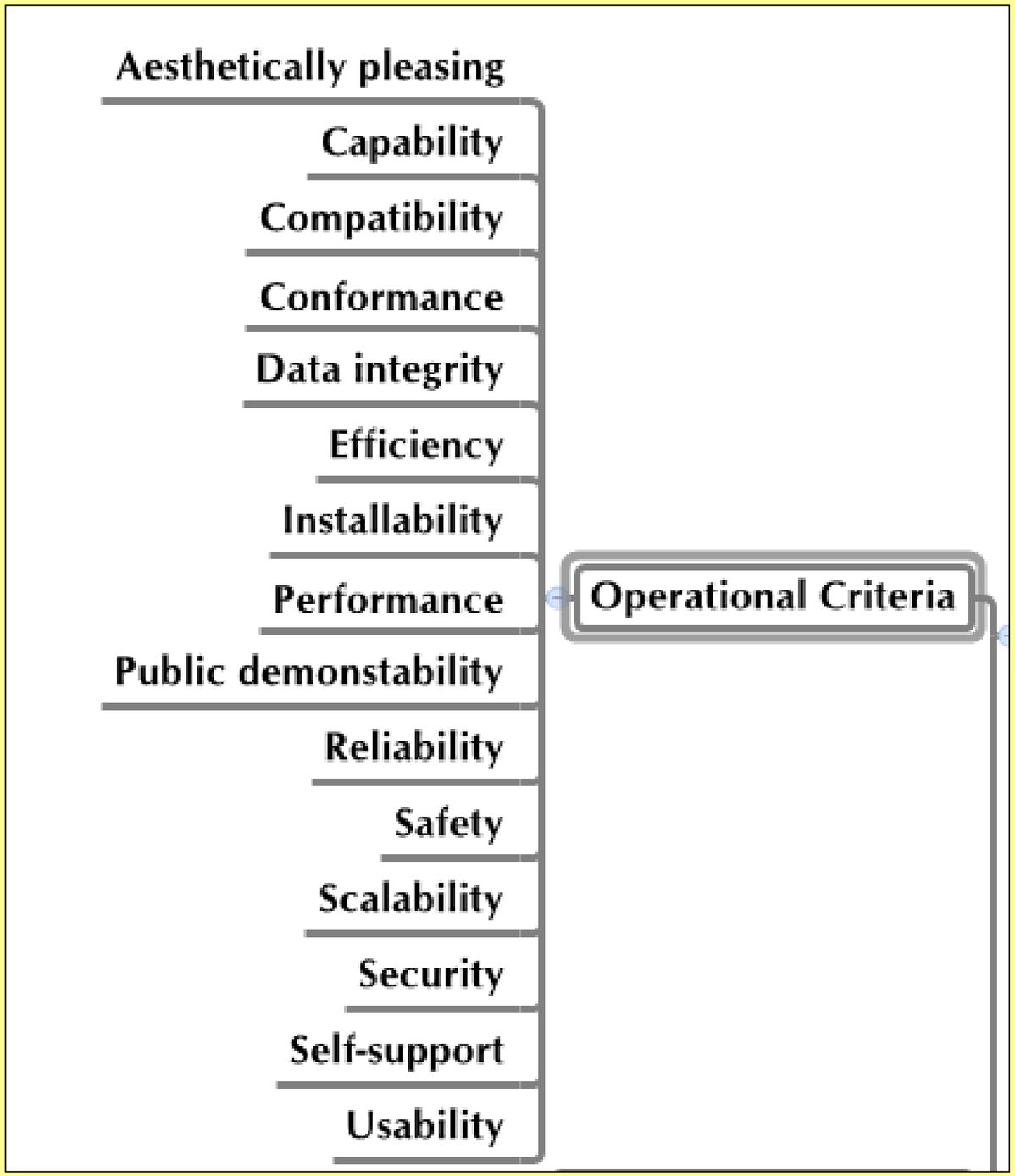
Quality criteria are particularly prone to variation across contexts:

- The more some criterion matters to you, the more finely you will analyze it.

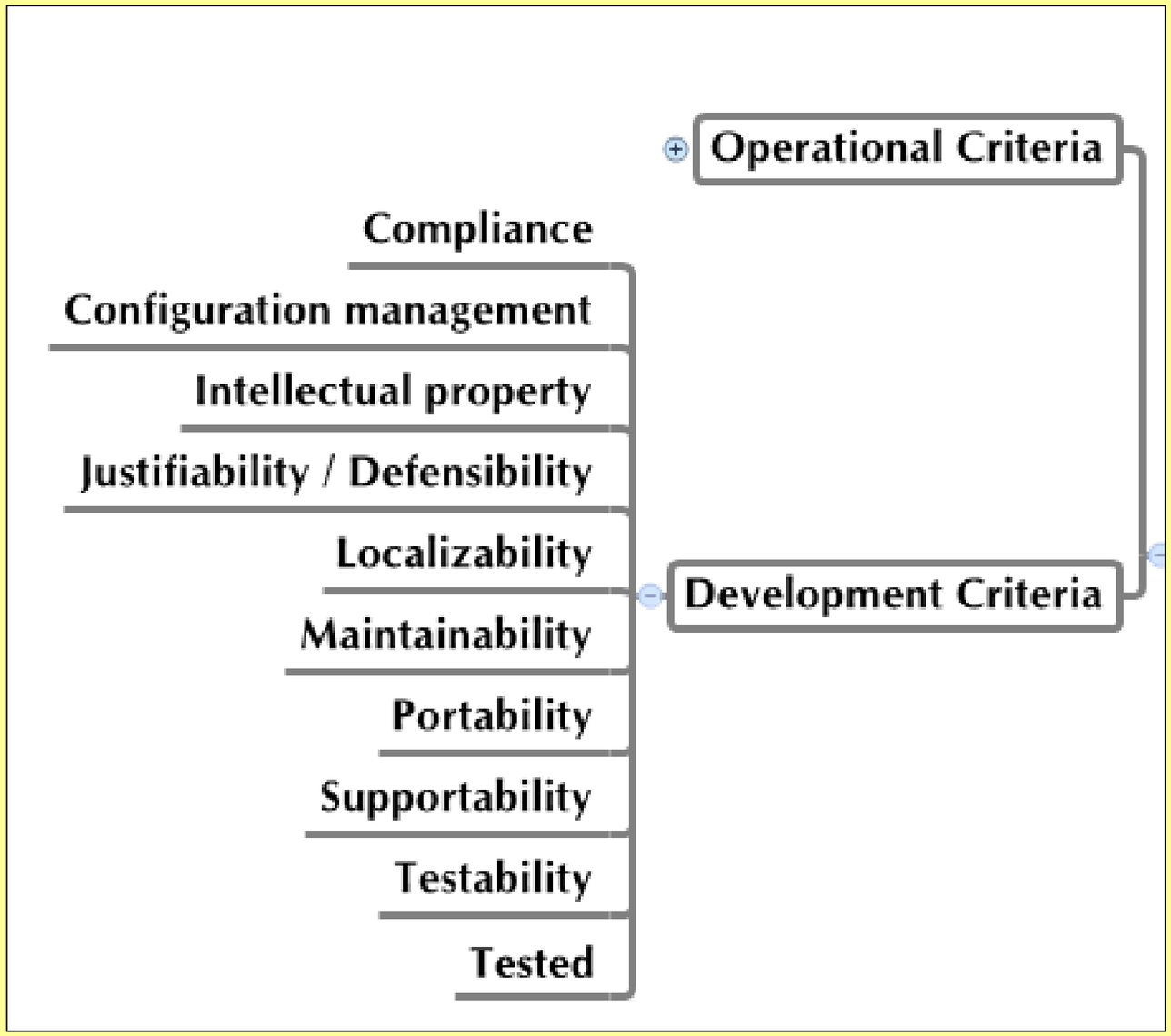


Emilsson, Jansson & Edgren present their customization in Software Quality Characteristics 1.0 at http://thetesteye.com/posters/TheTestEye_SoftwareQualityCharacteristics.pdf

Here's the next level down in my version of the model. It contains all of Bach's categories, but adds a few that are useful to me.



Most people who work seriously with this model customize it to meet their needs.



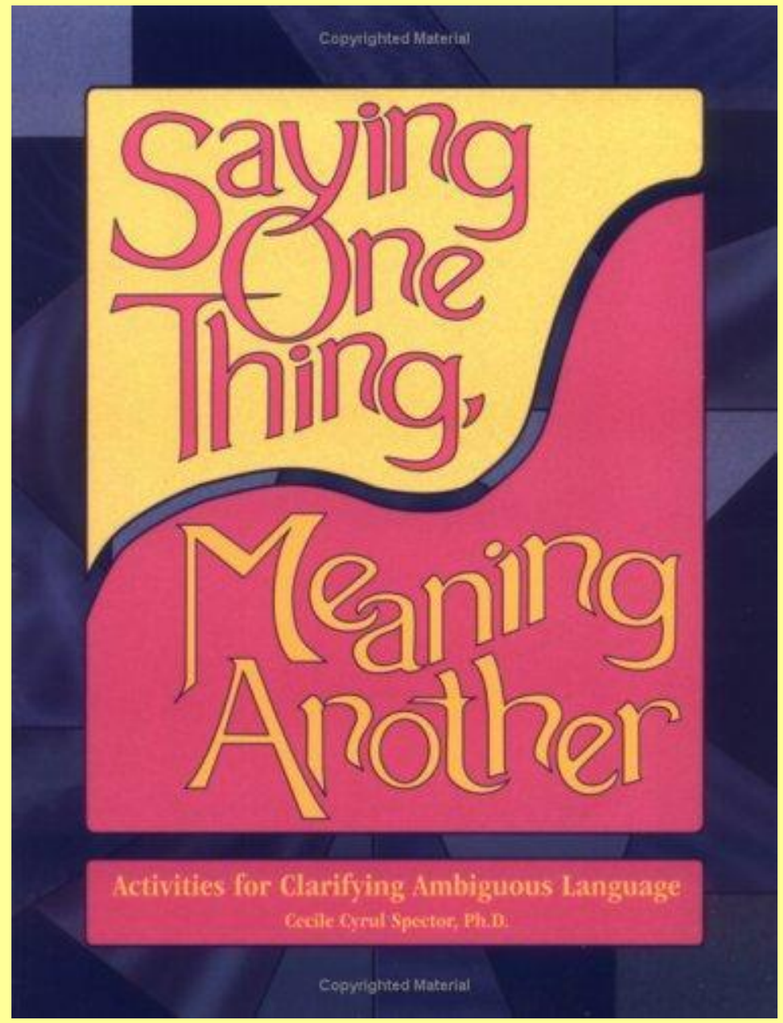
USING HTSM FOR ACTIVE READING

- Each statement of interest goes onto the map
- Add notes to include:
  - Test ideas
  - Special data values
  - Interactions with other variables
  - Why this item is important

We'll work through an example of this in the assignment.

CRITICAL QUESTIONS

1. What is the specification?
2. Why did they create it?
3. Who are the stakeholders?
4. What are you trying to learn or achieve with the spec?
5. What are the consequences of nonconformity?
6. What claims does the specification make?
7. What ambiguities must be resolved?
8. How should you use it to create tests?



Ambiguity analysis

Many sources of ambiguity in software design & development. A few examples:

- Wording or interpretation of specs or standards
- "Technical terms" have specific meanings to some readers but incompatible dictionary-meanings to most.
- Expected responses of the program to invalid or unusual inputs
- Behavior of undocumented features
- Conduct and standards of regulators / auditors
- Customers' interpretation of their needs and the needs of the users they represent
- Definitions of compatibility among 3rd party products

Whenever there is ambiguity, there is opportunity for a defect.

Many testers find Richard Bender's notes particularly helpful.

http://benderrbt.com/Ambiguityprocess.pdf

CRITICAL QUESTIONS

1. What is the specification?
2. Why did they create it?
3. Who are the stakeholders?
4. What are you trying to learn or achieve with the spec?
5. What are the consequences of nonconformity?
6. What claims does the specification make?
7. What ambiguities must be resolved?
8. How should you use it to create tests?

Driving tests from the spec

For every statement of fact in the specification:

- Create at least one test that tries to prove the statement false
- Create tests that vary the parameters of the statement (e.g. test boundary conditions)
- Create tests of the reasonable implications of the statement
- Create tests of this statement in conjunction with related statements
- Create tests of scenarios that apply the statement in the process of achieving a program benefit.

The level of depth you will choose should depend on the kinds of information you're looking for and the risks you're trying to manage.

Traceability matrix

	Item 1	Item 2	Item 3	Item 4	Item 5
Test 1	X	X	X		
Test 2		X		X	
Test 3	X		X	X	
Test 4			X	X	
Test 5				X	X
Totals	2	2	3	4	1

A traceability matrix maps tests to test items. For each test item, you can trace back to the tests that test it.

Traceability matrix

- Useful for tracking specification coverage
- Each test item in its own column.
  - A test item is anything that must be tested: might be a function, a variable, an assertion in a specification, a device that must be tested.
- One row per test case.
- A cell shows that this test tests that test item.
- If a feature changes, you can quickly see which tests must be reanalyzed, probably rewritten.
- In general, you can trace back from a given item of interest to the tests that cover it.
- This doesn't specify the tests, it merely maps their coverage.

Review

- Specification-based testing
  - Discovering what the specification is and what it says is probably the hardest task
    - Implicit, explicit specifications
    - HTSM as an active reading tool
  - Important to analyze the content and context of the specification
  - Spec-driven testing can be done at many levels of thoroughness. Simple checking is common, but it won't tell you much.

End of Lecture 3

Black Box Software Testing: Introduction to Test Design: Lecture 4: Scenario Testing

Cem Kaner, J.D., Ph.D.
Professor of Software Engineering: Florida Tech

Rebecca L. Fiedler, M.B.A., Ph.D.
President: Kaner, Fiedler & Associates

These notes are partially based on research that was supported by NSF Grants EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers" and CCLI-0717613 "Adaptation & Implementation of an Activity-Based Online or Hybrid Course in Software Testing." 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.

OVERVIEW OF THE COURSE

	Technique	Context / Evaluation
1	Function testing & tours.	A taxonomy of test techniques.
2	Risk-based testing, failure mode analysis and quicktests	Testing strategy. Introducing the Heuristic Test Strategy Model.
3	Specification-based testing.	(… work on your assignment …)
4	Use cases and scenarios.	Comparatively evaluating techniques.
5	Domain testing: traditional and risk-based	When you enter data, any part of the program that uses that data is a risk. Are you designing for that?
6	Testing combinations of independent and interacting variables.	Combinatorial, scenario-based, risk-based and logical-implication analyses of multiple variables.

LECTURE 4 READINGS

Required reading

- Bolton, Michael (2007). Why we do scenario testing. http://www.developsense.com/blog/2010/05/why-we-do-scenario-testing/
- Carroll, John M. (1999). Five reasons for scenario-based design. Proceedings of the 32nd Hawaii International Conference on System Sciences, http://www.massey.ac.nz/~hryu/157.757/Scenario.pdf
- Kaner, Cem (2003). An introduction to scenario testing. http://www.kaner.com/pdfs/ScenarioIntroVer4.pdf
- Kaner, C. (2003). What is a good test case? http://www.kaner.com/pdfs/GoodTest.pdf

Recommended reading

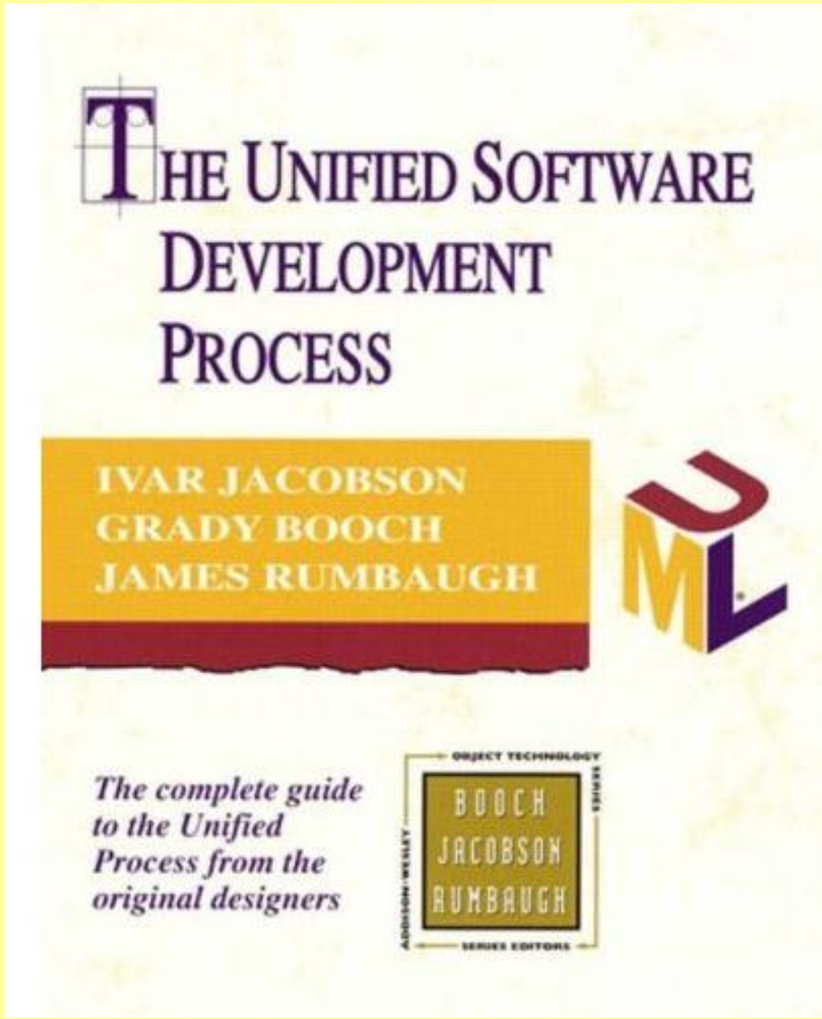
- Buwalda, Hans (2000b) "Soap Opera Testing," presented at International Software Quality Week Europe conference, Brussels. http://www.logigear.com/resource-center/software-testing-articles-by-logigear-staff/246-soap-opera-testing.html
- Charles, Fiona A. (2009). Modeling scenarios using data. STP Magazine. http://www.quality-intelligence.com/articles/Modelling%20Scenarios%20Using%20Data_Paper_Fiona%20Charles_CAST%202009_Final.pdf
- Collard, R. (1999, July) “Developing test cases from use cases”, Software Testing & Quality Engineering, available at www.stickyminds.com
- Hackos, J.T. & Redish, J.C. (1998). User and Task Analysis for Interface Design. Wiley

SCENARIOS FOR BEGINNERS

"Use case" is a popular and influential design idea in software engineering.

"A use cases specifies a sequence of actions, including variants, that the system can perform and that yields an observable result of value to a particular actor."

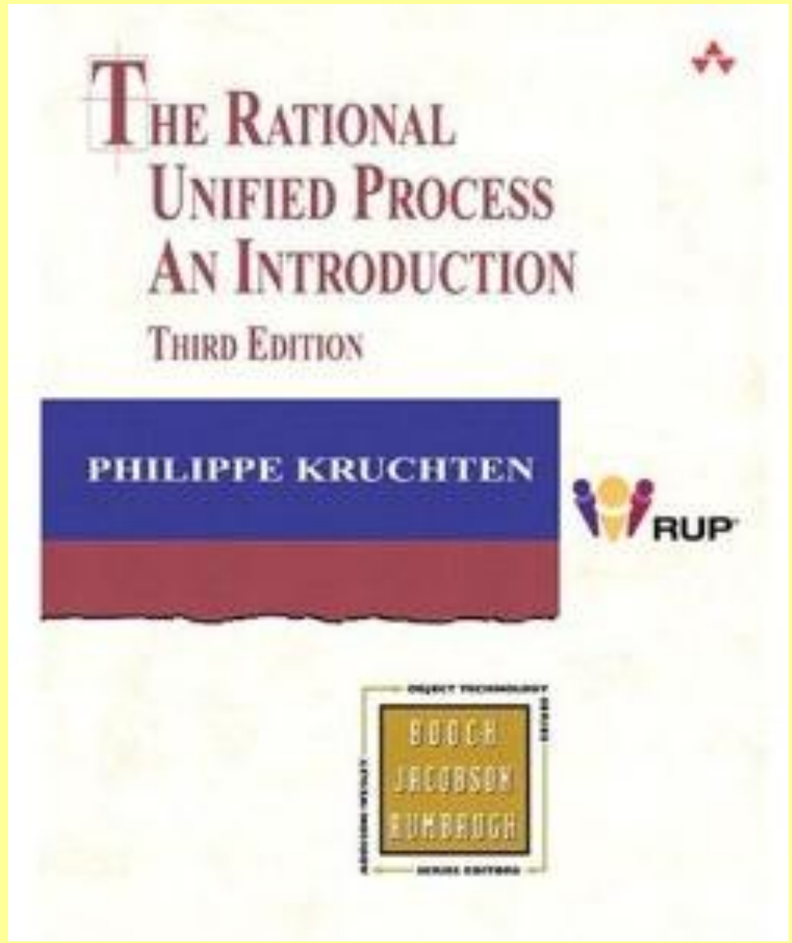
Jacobson et al, p. 41



SCENARIOS FOR BEGINNERS

Concepts within the use case:

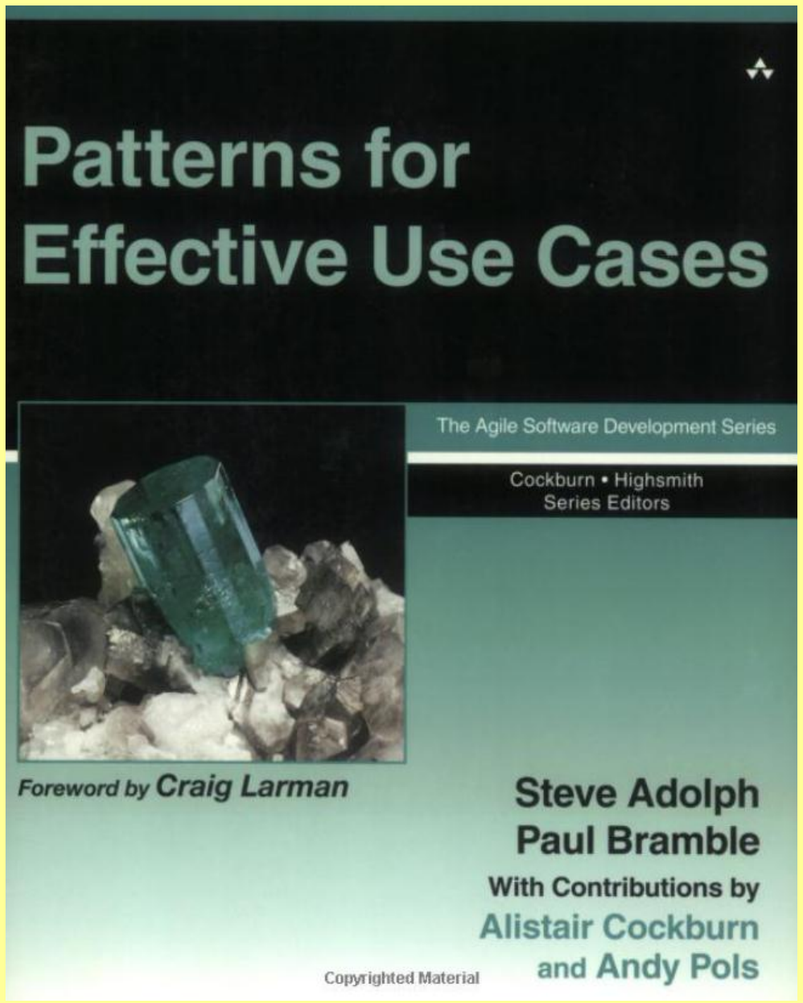
- **Actor:** a person, process or external system that interacts with your product.
- **Action:** "An action results in a change of state and is realized by sending a message to an object or modifying a value in an attribute." (Jacobson et al, 426). (*Something the actor does as part of the effort to achieve the goal*.)
- **Goal.** The goal is to reach a desired state of the system (the observable result of value).



SCENARIOS FOR BEGINNERS

Concepts within the use case:

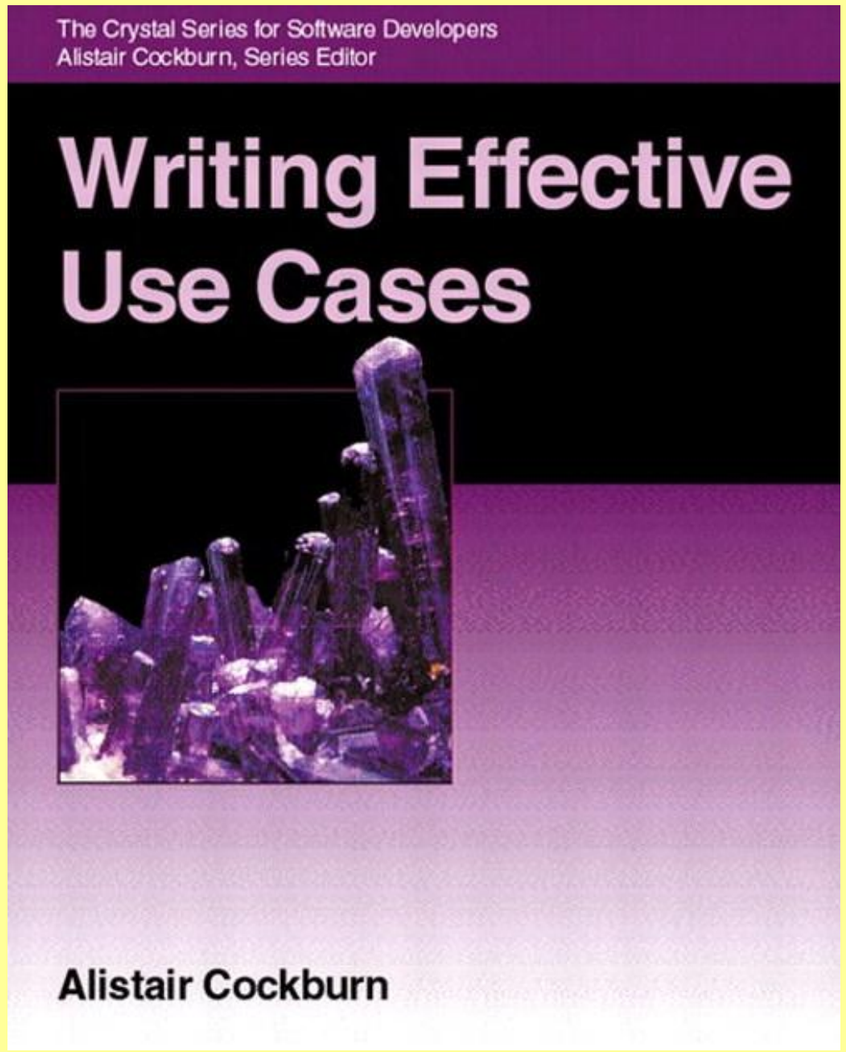
- **Sequences of actions: "**A specific flow of events through the system. Many different flows are possible and many of them may be very similar. To make a use-case model understandable, we group similar flows of events into a single use case." Krutchen (2003)
- **Sequence diagram:** A diagram that shows actions and states of a use case, emphasizing the ordering of the actions in time.



SCENARIOS FOR BEGINNERS

- Brainstorm and list the primary actors
- Brainstorm and exhaustively list user goals for the system
- Capture the summary goals (higher-level goals, which include several sub-goals. These capture the meaningful benefits offered by the system).
- Select one use case to expand.
- Capture stakeholders and interests, preconditions and guarantees.
- Write the main success scenario.
- Brainstorm and exhaustively list extension conditions (such as alternate sequences to achieve the same result, or sequences that lead to failure.)

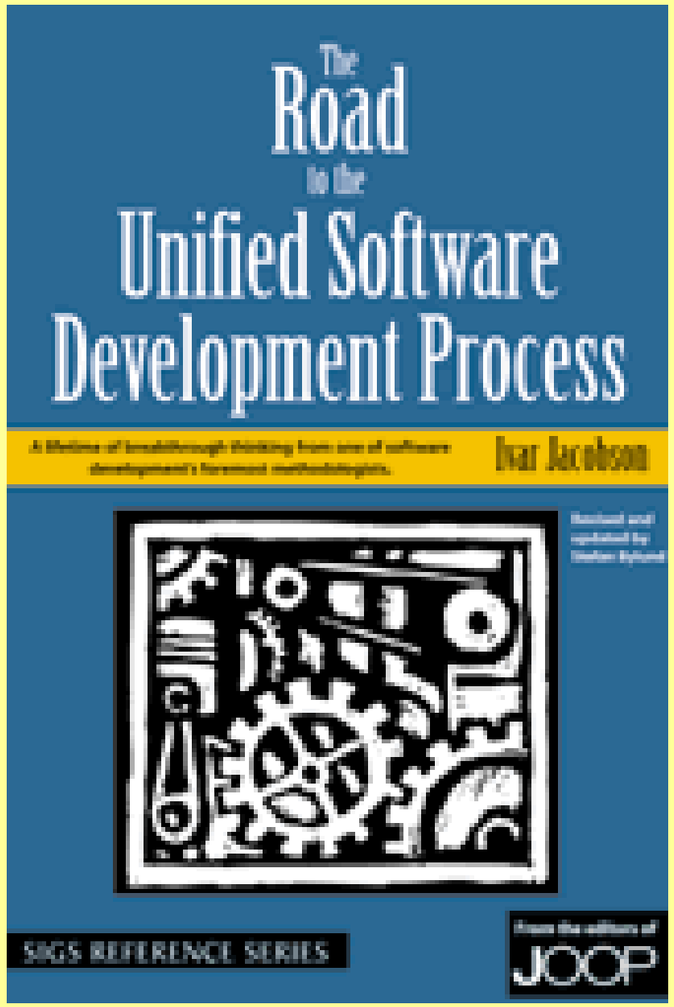
Taken / summarized from Cockburn (2001).



SCENARIOS FOR BEGINNERS

The Rational Unified Process defines scenarios in terms of use cases. **Under their definition**:

- A scenario is an instantiation of a use case (specify the values of the use case's data to create one instance of the use case)
- A RUP-scenario traces one of the paths through the use case. If you actually execute the path, you are running a scenario test (See Collard, 1999).
- Thorough use-case-based testing involves tracing through all (most) of the paths through all (most) of the use cases, paying special attention to failure cases.



Benefits of use-case based testing

Encourages the tester to:

- Identify the actors in the system
  - Human
  - Other processes or systems
- Inventory the possible actor goals
- Identify the benefits of the system (via identifying the summary goals)
- Develop some method (sequence diagrams, outlines, textual descriptions, whatever) for describing a sequence of actions and system responses that ultimately lead to a result.
- Develop variations of a basic sequence, to create meaningful new tests.

Incompetent use-case modelers consider only the happy paths ("main success scenarios") or simplistic deviations from them. This is common, but foolish.

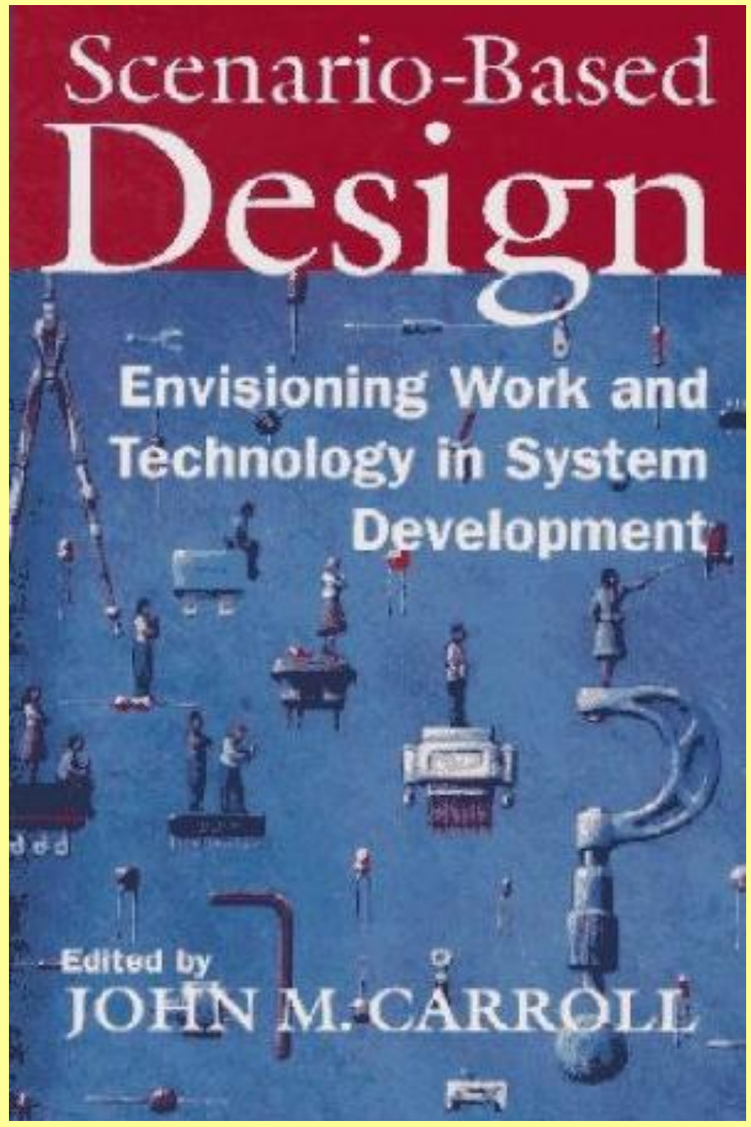
Evaluating this approach

- A use-case based approach to testing provides a good starting point if you don't know much about the application.
  - Provides a structure for tracing through the application
  - As simple as function testing but works several functions together
- The Rational Unified Process, and the concept generally of use cases, have been widely adopted in the academic community and (especially use cases) in the agile development community.
  - Atif Memon has published an interesting line of research on automated development of scenario paths.

This basic approach to scenario testing is easier for students because a course gives students little time to develop a deep appreciation of the software under test.

EVALUATING THIS APPROACH

- This approach abstracts out the human element:
  - Because the actor may not be human, actors are described in ways that are equally suitable for things that have no consciousness.
  - Human goals go beyond a desired program state. They are more complex.
  - In humans, goals are intimately connected with motivation—Why does this person want to achieve this goal? How important is it to them? Why?
  - In humans, failure to achieve a goal causes consequences, including emotions. How upset will the user be if this fails? Why?



EVALUATING THIS APPROACH

- There **can be** scenarios with no people in them, but when there are people, scenario writers are interested in them.
- Even if all the obvious actors are human, there is a person who has started the scenario in motion. The scenario analyst will be on the lookout for this human-in-the-background and will be interested in the motivation and reactions of that person.
- More generally, what I know as "scenarios" involves a much richer view of the system and the people who use it, including details that use-case authors would normally exclude (see Cockburn's recommendations on what to include and what to abstract out).

Even though use-case based testing is useful in its own right, *as a basic approach to scenario testing*, it misses the deep value of what we know as scenario analysis.

THE SCENARIO CONCEPT

- Early scenarios:
  - Imagine a hypothetical future event or crisis
    - What effects or side-effects is it likely to have?
    - How will existing systems or policies deal with it?
- Alexander & Malden (2004); Kahn (1967); Wack (1985a); Walker (1994); Wikipedia: "Scenario planning"

Kahn's list of benefits of scenario-based thinking

- Call attention to the larger range of possibilities that must be considered in the analysis of the future
- Dramatize and illustrate the possibilities
- Force analysts to deal with details and dynamics that they might avoid if they focus on abstract considerations
- Illuminate interactions of psychological, social, economic, cultural, political, and military factors, including the influence of individual personalities … in a form that permits the comprehension of many interacting elements at once.
- Consider alternative possible outcomes of certain real past and present events

abstracted from Kahn (1967), pp. 262-264

Exemplars from other fields

Here are a couple of other papers that you might find interesting. These illustrate scenario-based planning in other fields.

- Alexander (2000): Reports on using scenarios to teach principles of emergency planning and management
- Rippel & Teply (2008): Reports on using scenarios to test banks' ability to withstand stressors (what they call "risk events")

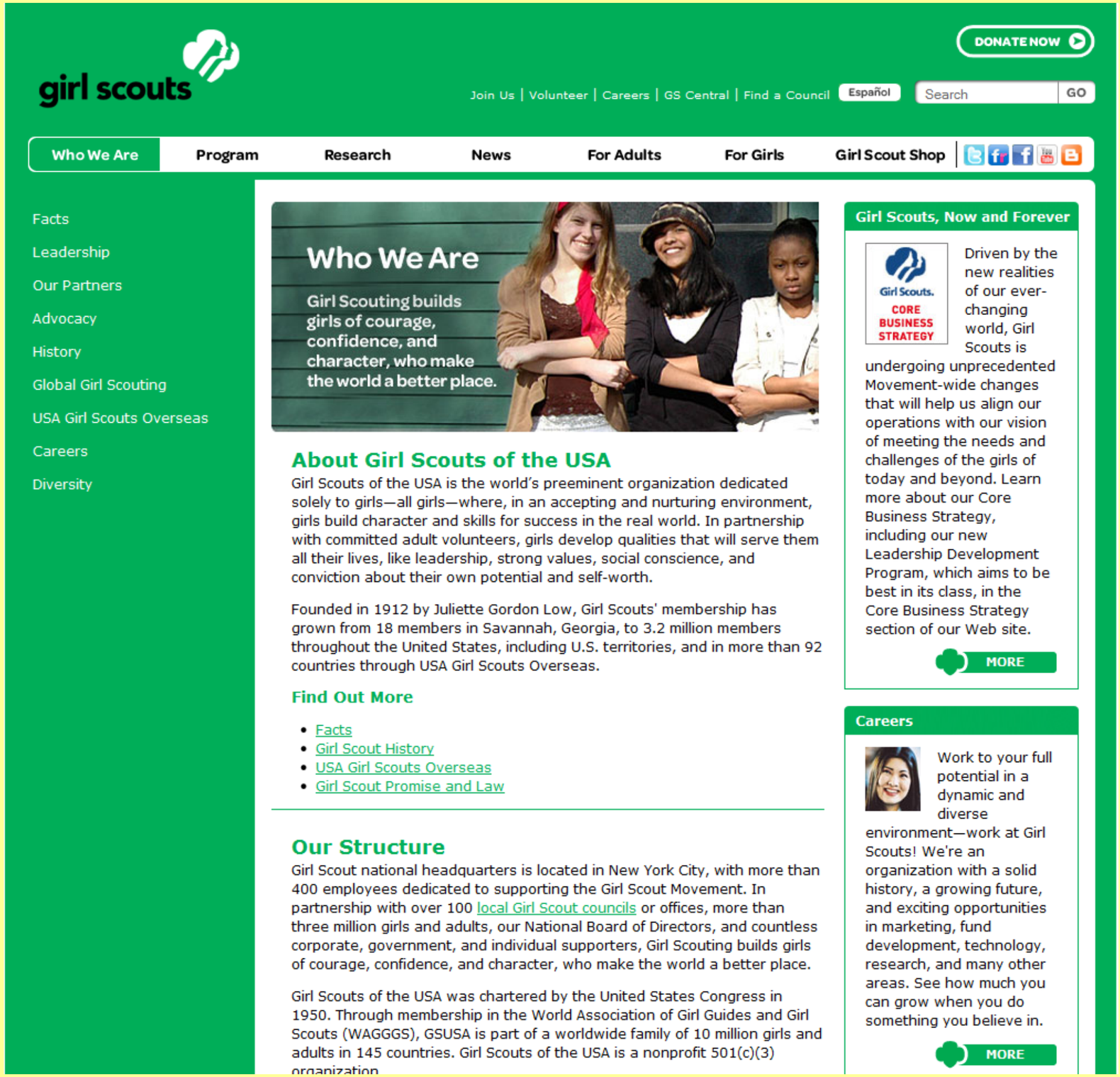
Bounce!

THE POSTAGE STAMP BUG

Postage Stamp Bug

Scenario 1:

A real user wants the program to place the logo in exactly the place that the logo should be.



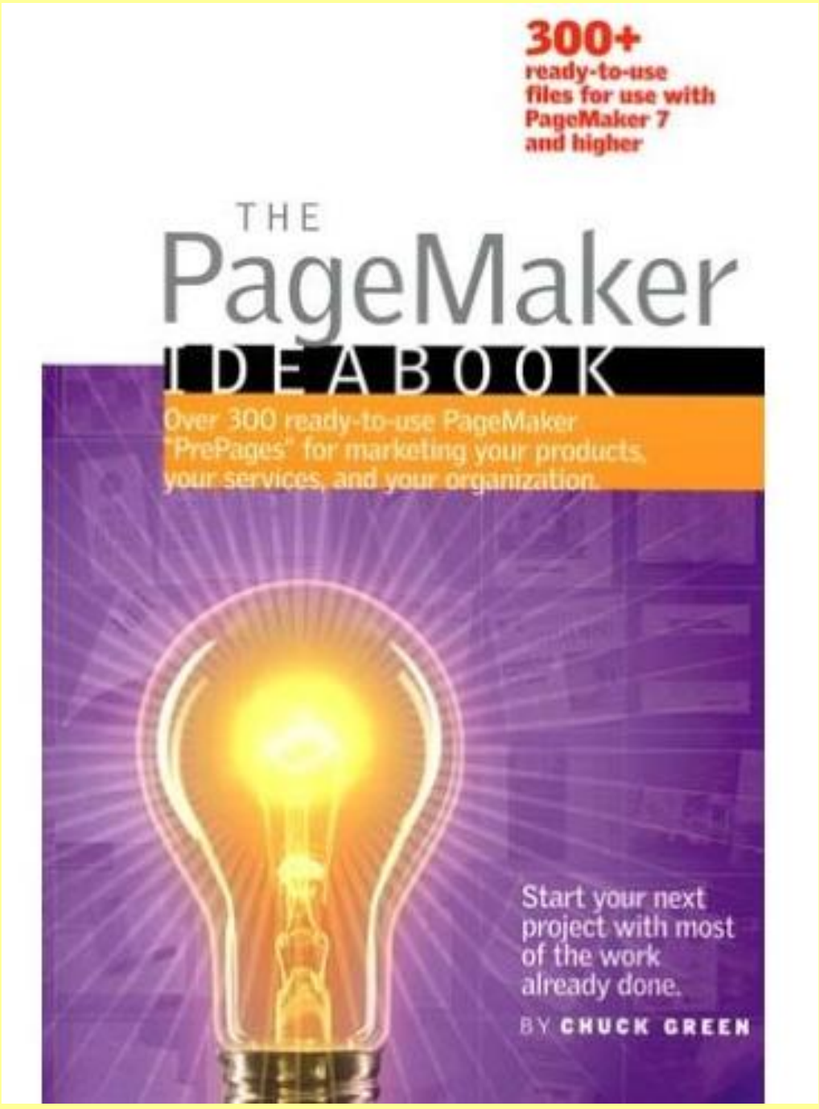
http://www.girlscouts.org/who_we_are/

Postage Stamp Bug

Scenario 2:

- Create designs that copy PageMaker templates
- Surprise!

  (*Some of the designs need a graphic pasted at the postage stamp bug location.)*



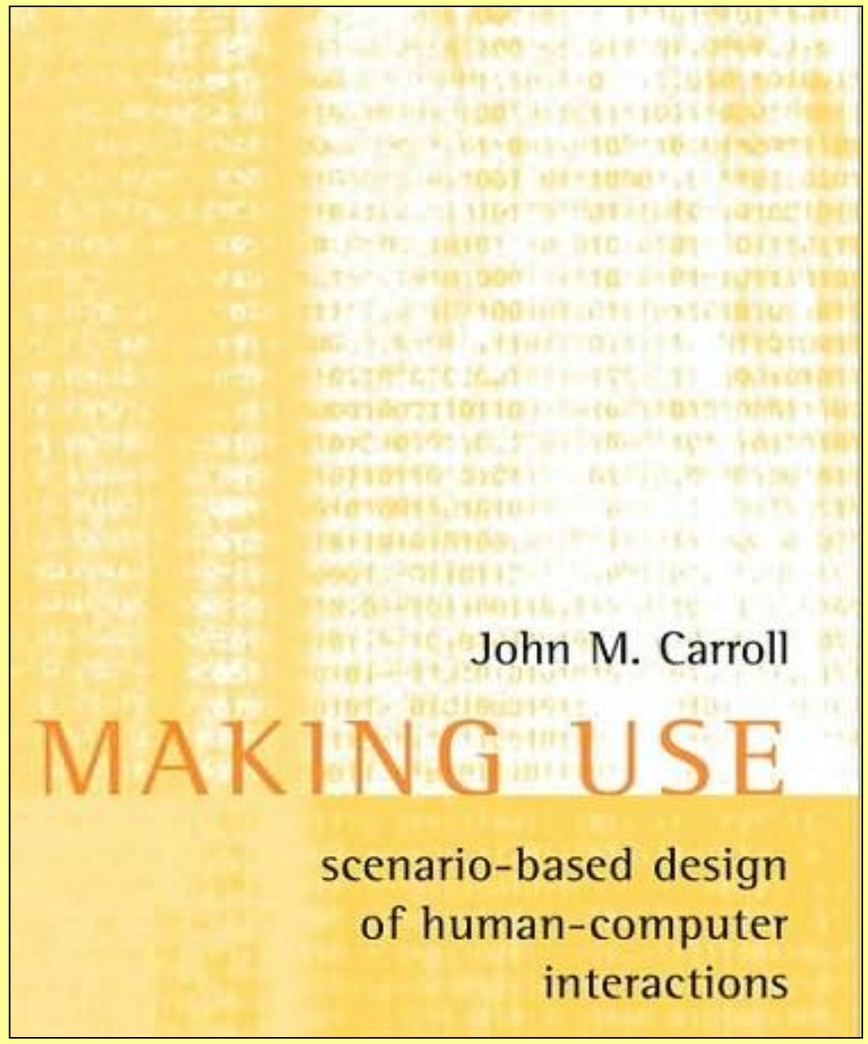
THE SOFTWARE SCENARIO

A **scenario** is a coherent story about how someone uses (or tries to use) the program.

A **scenario test** uses a scenario as a tool for evaluating a program's behavior

The elements of the story (adapted from Carroll, 1999)

- Setting
- Agents or actors
- Goals or objectives
- Motivations and emotions
- Plot (sequences of actions and events)
- Actions & events can change the goals



Attributes of scenario tests

Ideal scenario test has several characteristics:

- The test is **based on a coherent story** about how the program is used, including goals and emotions of people.
- The **story is credible**. Stakeholders will believe that something like it **probably will happen**.
- The **story is motivating**. A stakeholder with influence will advocate for fixing a program that failed this test.
- The story involves **complexity**: a complex use of the program or a complex environment or a complex set of data.
- Test results are **easy to evaluate**. This is important for scenarios because they are complex.

What testers learn from scenarios

Many test techniques tell you how the program will behave in the first few days that someone uses it.

- Good scenario tests go beyond the simple uses of the program to ask whether the program is delivering the benefits it should deliver
- Good scenarios often give you insight into frustrations that an experienced user will face—someone who has used the program for a few months and is now trying to do significant work with the program.

Approaches to combination testing

- ***Mechanical (or procedural).*** The tester uses a routine procedure to determine a good set of tests
- ***Risk-based.*** The tester combines test values (the values of each variable) based on perceived risks associated with noteworthy combinations
- ***Scenario-based.*** The tester combines test values on the basis of interesting stories created for the combinations

Scenario-based thinking provides a strategy for selecting meaningful combinations, such as combinations important to the experienced user.

17 LINES OF INQUIRY FOR SUITES OF SCENARIOS

1. Create follow-up tests for bugs that look controversial or deferrable.

The postage stamp bug is an example of this kind of scenario.

This is NOT the most important kind of scenario.

The next 16 start from ideas about the user, the program, the task, or the market. They are the basis for SUITES of scenarios rather than one scenario focused on one bug.

Consider a hypothetical example

Imagine the **GetAJob** product.

It helps the user:

- Create resumes
- Print business cards
- Mine job openings from the web
- Enter data into standard job application forms on the web
- Track contacts with employers
- Track contacts with recruiters
- Track job-seeking expenses
- etc.

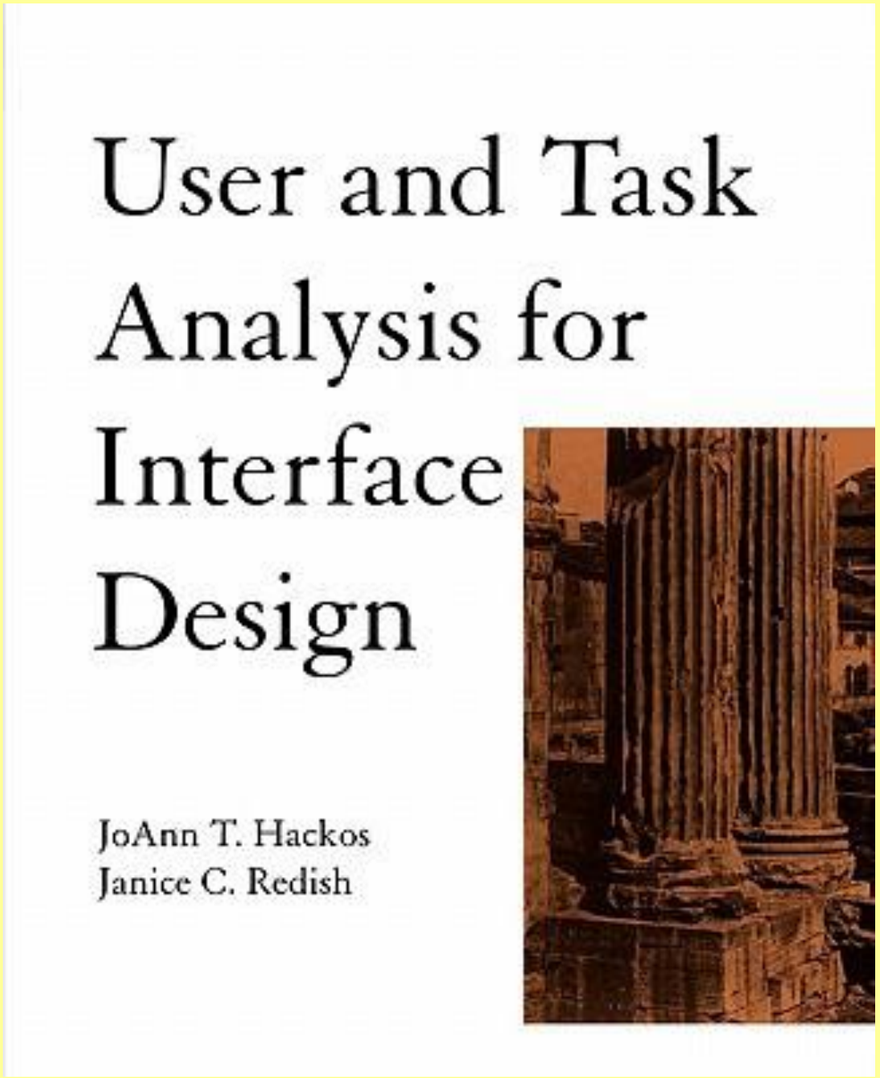
17 LINES OF INQUIRY FOR SUITES OF SCENARIOS

2. List possible users. Analyze their interests and objectives.
   - **Interests**: broader motivators of the person.
   - **Objectives**: Specific tasks the user wants to achieve with the program.
   - Examples:
     - The traditionalist
     - The young networker
     - The socialite salesperson
     - The support group

17 LINES OF INQUIRY FOR SUITES OF SCENARIOS

3. Work alongside users to see how they work and what they do.
   - What are they doing? Why?
   - What confuses them?
   - What irritates them?

– All of those become tests…



17 LINES OF INQUIRY FOR SUITES OF SCENARIOS

4. Interview users about famous challenges and failures of the old system.

For more on this (and all of the other lines of inquiry that study users or workflows), see the readings on task analysis.

17 LINES OF INQUIRY FOR SUITES OF SCENARIOS

5. Look at the specific transactions that people try to complete, such as opening a bank account or sending a message.

   You can design scenarios (one, or probably more) for each transaction, plus scenarios for larger tasks that are composed of several transactions.

Transaction processing systems:

http://en.wikipedia.org/wiki/Transaction_processing

17 LINES OF INQUIRY FOR SUITES OF SCENARIOS

6. Work with sequences

- People (or the system) typically do tasks (like Task X) in an order. What are the most common orders (sequences) of subtasks in achieving X?
- It might be useful to map Task X with a behavior diagram.

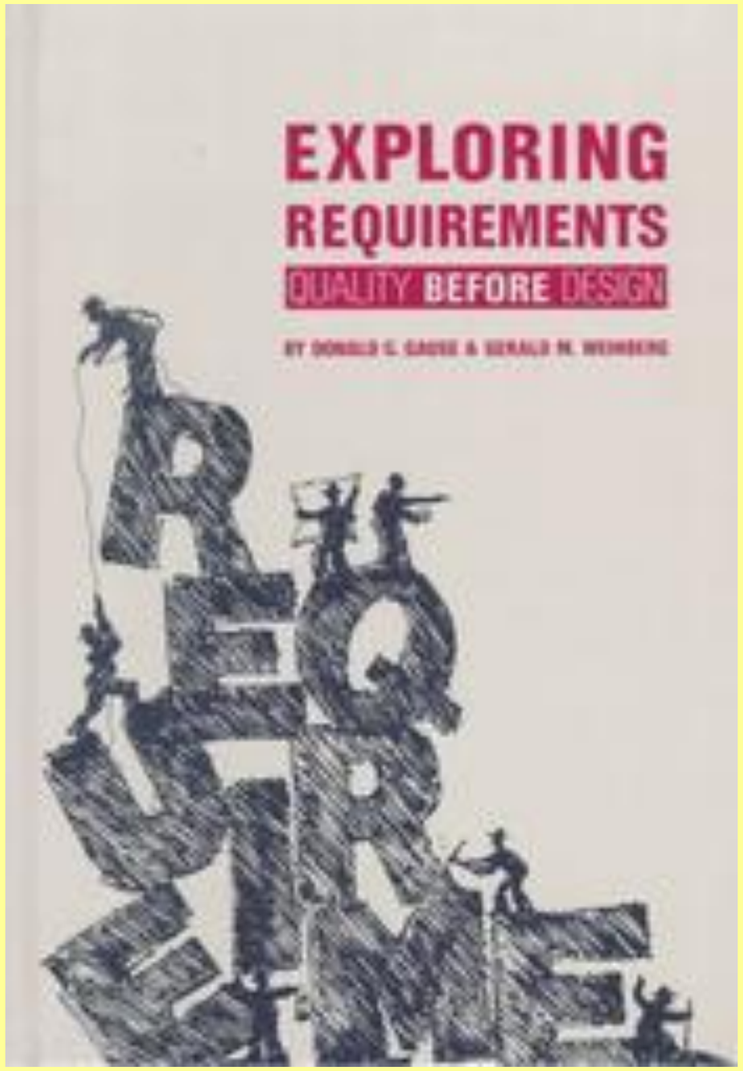
This is the closest analog to the use-case based scenario. But feel free to consider motivation and consequence, not just the goal and the alternate sequences.

17 LINES OF INQUIRY FOR SUITES OF SCENARIOS

7. Consider disfavored users.

   How do they want to abuse your system? Analyze their interests, objectives, capabilities, and potential opportunities.

Gause & Weinberg discuss disfavored users in *Exploring Requirements*.



17 LINES OF INQUIRY FOR SUITES OF SCENARIOS

8. What forms do the users work with? Work with them (read, write, modify, etc.)
   - GetAJob probably:
     - Offers several standard resume templates
     - Automatically fills in fields in employer-site or recruiter-site forms

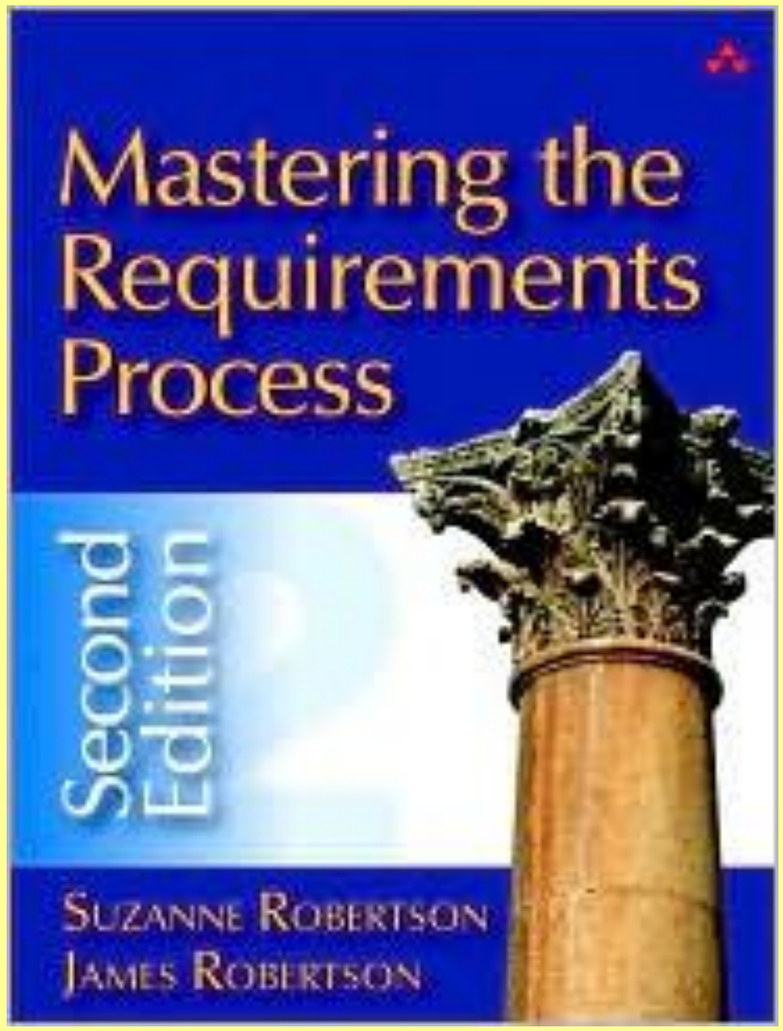
17 LINES OF INQUIRY

9. Write life histories for objects in the system.
   - How was the object created, what happens to it, how is it used or modified, what does it interact with, when is it destroyed or discarded?
   - GetAJob, for example, includes:
     - Resumes
     - Contacts
     - Downloaded ads
     - Links to network sites
     - Emails

Just as you can create a list of possible users and base your scenarios on who they are and what they do with the system, you can create a list of objects and base your scenarios on what they are, why they're used, and what the system can do with them.

17 LINES OF INQUIRY

10. List system events. How does the system handle them?
    - An event is any occurrence that the system is designed to respond to.
        - Business events, such as going to an interview, sending a resume, getting called by a prospective employer. (Robertson & Robertson are helpful for identifying these)
        - Anything that causes an interrupt is an event the system has to respond to



17 LINES OF INQUIRY

11. List special events. The system might change how it works or do special processing in the context of this event.
    - Predictable but unusual occurrences
    - Examples:
        - Last (first) day of the quarter or of the fiscal or calendar year
        - While you installing or upgrading the software
        - Holidays

17 LINES OF INQUIRY

12. List benefits and create end-to-end tasks to check them.
    - What benefits is the system ***supposed to*** provide?
    - For example, if the current project is an upgrade, what benefits will the upgrade bring?
    - Don't rely only on an official list of benefits. Ask stakeholders what they *think* the benefits of the system are supposed to be.
    - Look for misunderstandings and conflicts among the stakeholders.

17 LINES OF INQUIRY

13. Work with competing systems, or read books / articles about what systems like this are supposed to do
    - What programs compete with the GetAJob system? How do they work? If you knew them well, what would you expect of GetAJob?
    - What programs offer some of the capabilities of GetAJob? For example, if you knew contact management programs well, what expectations would you have of GetAJob's contact management features?

17 LINES OF INQUIRY

14. Study complaints about this system's predecessor or competitors.
    - Software vendors usually create a database of customer complaints.
    - Companies that write software for their own use often have an in-house help desk that keeps records of user problems.
    - Look for complaints about your product or similar ones online.
    - Read the complaints. Take "user errors" seriously—they reflect ways that the users expected the system to work, or things they expected the system to do.

17 LINES OF INQUIRY

15. Create a mock business. Treat it as real and process its data.
    - Choose the characteristics of the business well. Simulate a business that fits the profile of your intended users.
    - Create events that are realistic for that business, and see how your system copes with them.
    - When you run into problems or limitations, consider (and describe) how they impact your simulated business.

17 LINES OF INQUIRY

16. Try converting real-life data from a competing or predecessor application.
    - Test GetAJob 3.0 by feeding it user files from previous versions
        - Does it remember all the contacts?
        - Does it look up the right receipts and do the right calculations for job-hunting tax deductions?
        - How can you tell?

This is a historically common way to test a program, but many people use it without a clear oracle. How will you recognize an error, such as one that is formatted correctly but the number is wrong?

17 LINES OF INQUIRY

17. Look at the output that competing applications can create.
    - How would you create these reports / displays / export files / whatever in your application?

To create a suite of scenarios:

- Wiser to design a collection of scenarios by following one line of inquiry at a time than by combining them.
- For example, list of types of objects in the system (so that you can develop a set of possible life histories for each)
- Given an item in the list, ask scenario-building questions
- Do this for several scenarios
  - Can build several scenarios for each item (type of object) in the list

To create a suite of scenarios:

Given an item in the list, ask the scenario questions:

- **How to create a story that people will listen to?**
  - Setting
  - Agents or actors
  - Goals or objectives
  - Plot (sequences of actions and events)
  - Actions and events can change goals
  - Emotions
- **Note: the expected result of the story is the result you expect if the program is working correctly**.

- **Coherent story**
- **Credible**
- **Motivating**
- **Complex**
- **Easy to evaluate**

To create a suite of scenarios:

Ask the scenario questions:

- **What would make a story about this be credible?**
  - When would this come up, or be used?
  - Who would use it?
  - What would they be trying to achieve?
  - Competitor examples?
  - Spec / support / history examples?

- **Coherent story**
- **Credible**
- **Motivating**
- **Complex**
- **Easy to evaluate**

To create a suite of scenarios:

Given an item in the list, ask scenario-building questions:

- **What is important (motivating) about this?**
  - Why do people care about it?
  - Who would care about it?
  - What does it relate to that modifies its importance?
  - What gets impacted if it fails?
  - What does failure look like?
  - What are the consequences of failure?
    - Does it ever take on urgency?

- **Coherent story**
- **Credible**
- **Motivating**
- **Complex**
- **Easy to evaluate**

Motivating scenarios

- Scenarios are powerful tools for building a case that a bug should be fixed
  - Makes the problem report meaningful to a powerful stakeholder who should care about this particular failure
- Inability to develop a strong scenario around a failure may be a signal that the failure is not well understood or not important.

To create a suite of scenarios:

Given an item in the list, ask the scenario questions:

- **How to increase complexity?**
  - What does this naturally combine with?
  - What benefits involve this and what collection of things would be required to achieve each?
  - Can you make it bigger? Do it more? Work with richer data? (What boundaries are involved?)
  - Will any effects of the scenario persist, affecting later behavior of the program?

- **Coherent story**
- **Credible**
- **Motivating**
- **Complex**
- **Easy to evaluate**

SCENARIO COMPLEXITY

- Test each feature in isolation (or in small mechanical clusters of features) before testing it inside scenarios.
  - Reach straightforward failures sooner and more cheaply
  - If you keep and reuse tests, it is better to expose weak designs with cheap function tests than more expensive scenarios
  - Combination failures are harder to troubleshoot. Simple failures that appear first inside a combination can be unnecessarily expensive to troubleshoot
  - Scenarios are prone to blocking bugs: a broken feature blocks running the rest of the test. Once that feature is fixed, the next broken feature blocks the test.
- Adding complexity arbitrarily won't work. The story must still be coherent and credible.

Scenario complexity

Scenario testing provides **one approach** to designing tests that combine several variables or sequences of operations.

- Mechanical
  - Combinations that you can generate by following a routine procedure.
- Risk-based
  - Combinations that are perceived as more likely to yield failure or yield consequences that are more serious if failure occurs
- Scenario-based
  - **Combinations that can provide insight into the value of the product**

We'll return to combination tests in Lecture 6.

To create a suite of scenarios:

Given an item in the list, ask the scenario questions:

- **How to design an easy-to-evaluate test?**
  - Self-verifying data sets?
  - Automatable partial oracles?
  - Known, predicted result?
- **Evaluability is important because so many failures have been exposed by a good scenario but missed by the tester**

- **Coherent story**
- **Credible**
- **Motivating**
- **Complex**
- **Easy to evaluate**

Practical tips for describing the scenario

- Sketch the story, briefly. You don't have to write down the details of the setting and motivation if you understand them. (Add these details to your bug reports, as needed.)
  - Some skilled scenario testers add detail early. See Hans Buwalda's presentation on Soap Opera testing.
- Only write down the steps that are essential (essential = you will forget or your are likely to make a mistake)
- Your expected result is ALWAYS correct program behavior.

Scenarios & requirements analysis

- 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 decide or recommend how the product should work. Her task is to expose credible concerns to the stakeholders.
- The tester doesn't have to make design tradeoffs. Her task is to expose the consequences of those tradeoffs, especially consequences that are unanticipated or more serious 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.**

Designing scenario tests is much like doing a requirements analysis, but it is not requirements analysis.

They rely on similar information but use it differently.

COVERAGE

- In general, you cannot guarantee high ***code*** coverage from scenario testing.
- Each line of inquiry is like a tour.
  - You ***could*** explore that line thoroughly to achieve a level of coverage. Examples—cover many:
    - system events
    - objects created by the system
    - required benefits
    - features
  - However, coverage-oriented testing often uses simpler tests.

Reusing scenarios

- Documenting and reusing scenarios seems efficient because it takes work to create a good scenario.
- Scenarios often expose design errors but you soon learn what a test teaches about the design.
- Scenarios expose coding errors because they combine many features and much data. However, to cover more combinations, you need new tests, not repetition of old ones.
- It might be more effective to do regression testing using single-feature tests or unit tests, not scenarios.

Regression testing based on scenario tests might be less powerful and less efficient than regression based on other techniques.

A MORE GENERAL VIEW OF TEST DESIGN

GENERALIZING…

- We've now looked at
  - Function testing
  - Risk-based testing
  - Scenario testing
  - Specification-based testing
- And we've noted the existence of about another 70 techniques.
- **What sets techniques apart from each other?**
- **What makes a test a good or bad instance of a specific technique?**

TWO EXAMPLES OF TEST TECHNIQUES

Scenario testing

Tests are complex stories that capture how the program will be used in real-life situations.

These are combination tests, whose combinations are credible reflections of real use.

These tests are highly credible (stakeholders will believe users will do these things) and so failures are likely to be fixed.

Risk-based testing

Tests are derived from ideas about how the program could fail.

These tests might focus on individual variables or combinations.

These tests are designed for power and efficiency—find the bug quickly—rather than for credibility. Extreme-value tests that go beyond reasonable use are common.

Techniques differ in core attributes of "good" tests.

- Power
- Valid
- Value
- Credible
- Representative
- Non-redundant
- Motivating
- Performable
- Reusable
- Maintainable
- Information value
- Coverage
- Easy to evaluate
- Supports troubleshooting
- Appropriately complex
- Accountable
- Affordable
- Opportunity Cost

Most tests have these attributes to some degree. To evaluate a test, imagine possible tests that would have more of the attribute or less of it. Compared to those, where does this one stand?

POWER

A test is powerful if it is designed to be likely to expose a type of error.

- A test can be powerful even if it doesn't find a bug. The question is: *if the program has a bug of this type, will this test expose it?*
- A test can be powerful with respect to some types of bugs but weak with respect to others.
- **A more powerful test is more likely to expose a type of bug than a test that is less powerful for bugs of that kind.**

Validity

A test is **valid** if you can be sure that the problems it reveals are genuine problems.

- As an example of **invalidity**, imagine a failure that occurs only on a system that has insufficient memory (below the minimum-published requirements).
  - Some companies will treat this as a problem if the program doesn't fail gracefully.
  - Others will reject the failure, and the test, as unreasonable.

Value

A test has **value** if it reveals things that your clients want to know about the product or project.

- Low-value example: some companies treat corner cases as low value. They consider the extreme values so extreme that they don't care what happens if someone actually pushes the program to those limits.
- High-value example: Toys"R"Us lost a lot of money because their website couldn't handle high pre-Xmas volume. This was an extreme value that they would have wanted to know about, and that they probably spend a lot of money now to study.

CREDIBLE

A test is **credible** if the stakeholders will believe that people will actually do the things that were done in this test, or that events like the ones studied in this test are likely to happen.

- When someone says “no one would do that”, they are challenging the credibility of the test
- When someone says, “I don’t care what would happen if someone did this”, they are challenging the value of the test.

Representative

Call a test **representative** if it is focused on actions or events most likely to be tried or encountered by real users.

- A test can be credible but unrepresentative.
  - A test that emulates a situation that arises 0.05% of the time is credible but not very representative
  - A test that emulates a situation that arises every day is representative.

Non-redundant

Two tests can be similar in fundamental ways. For example, they might be focused on the same risk. They might rely on the same data or on values that are only trivially different.

A test technique is focused on **non-redundancy** if it selects one test from a group of similar ones and treats that test as a representative of the larger group.

Domain testing is an example of a technique that is focused on non-redundancy.

Motivating

A test is **motivating** if the stakeholders will want to fix problems exposed by this test.

- Motivating: A problem might be serious enough or potentially embarrassing enough that the company will want to fix it even if it is not credible (unlikely to ever arise in practice).
- Not motivating: A problem might be credible and valuable (the company is glad to know about it), but the company doesn't think it is important enough to fix. (Perhaps it documents the bug instead to facilitate later tech support.)

PERFORMABLE

A test is **performable** if the tester can do the test as designed.

- A manual test that requires the tester to type lots of data without making mistakes is not very performable. Nor is a test that requires the tester to do something at an exact time.
- You can often improve performability by storing difficult-to-enter data in files that can be loaded into the test or by automating some pieces of the test that are hard to do by hand.

Reusable

A test is **reusable** if it is easy and inexpensive to reuse it.

- Tests that are not very performable are not easily reused.
- However, a test can be highly performable today but hard to reuse because the program's design changes frequently (so reuse will require maintenance).

Maintainable

A test is **maintainable** if it is easy to revise in the face of product changes.

- Good maintainability is critically important for automated regression testing.
- Maintainability is irrelevant for many exploratory tests. If you don't intend to reuse it, you don't have to invest time making it maintainable.

INFORMATION VALUE

The **information value** of a test reflects the extent to which the test will increase your knowledge (reduce "uncertainty"), whether the program passes or fails the test.

- The question this asks is whether you are designing the test so that you will learn something of value whether the program passes or fails the test.
- Most regression tests have relatively little information value. They are more like demonstrations than like tests because no one expects them to expose many bugs. "Pass" teaches you almost nothing.

Karl Popper (e.g. *Conjectures & Refutations*) inspired our emphasis on the information value of tests. Boris Beizer describes the low information value of regression tests as the "Pesticide Paradox."

Information value

Exploratory software testing is

- a style of software testing that
- **emphasizes the personal freedom and responsibility of the individual tester**
- **to continually optimize the value of her work**
- by treating
  - test-related learning,
  - test design,
  - test execution, and
  - test result interpretation as
- mutually supportive activities that run in parallel throughout the project.

COVERAGE

Coverage measures the amount of testing of a given type that you have completed, compared to the population of possible tests of this type.

A test technique is **focused on** coverage if a designer using the technique could readily imagine a coverage measure related to the technique and would tend to create a set of tests that would have high coverage according to that measure.

No individual test has much coverage, but a group of tests can have high coverage.

Easy to evaluate

A test is **easy to evaluate** if the tester can determine easily and inexpensively whether the program passed or failed the test.

- Scenario tests are often hard to evaluate because the test creates a lot of output that has to be inspected by a human. (This is such a problem that we emphasize evaluability as a criterion for good scenarios.)

SUPPORTS TROUBLESHOOTING

A test **supports troubleshooting** if it provides useful information for the debugging programmer.

- Long-sequence tests must be very carefully designed to support troubleshooting. When a test fails after 10 hours of execution of a long sequence, it can be very hard to figure out what went wrong, when.
- Programs often output event logs that provide diagnostic information about unusual or undesirable events. These illustrate ways that the software under test can make tests more or less effective at supporting troubleshooting.

Appropriately complex

The design objective is that you should use more complex tests as a program gets more stable.

- Early in testing, complex tests are almost impossible to run. You will waste time trying to run complex tests before the program is stable enough to handle them.
- Later, you can finally run tests that realistically reflect the ways that experienced users will drive the program.

Accountable

A test is **accountable** if you can explain what you did, justify why you did it, and provide that you actually conducted the test.

- Accountability is often critical for companies whose tests are audited or otherwise likely to be inspected by regulators or in court.
- Accountability can be very costly, and the cost of it can drive people to rely on regression tests (old, documented tests) rather than inventing new ones.

Session-based test management is a popular method for improving the accountability of exploratory testing.

AFFORDABILITY

The **cost of a test** includes time and effort associated with it as well as its directly financial costs.

As an attribute of a test technique, affordability is concerned with:

- The absolute cost of testing in this way
- Whether you could find this information more cheaply (more efficiently).

A technique is more affordable if it is designed to reveal better information at the same cost or equivalent information at a lower cost.

OPPORTUNITY COST

Because you have an infinite number of potential tests, and therefore an infinite number of potential test-related task, every test and every task has opportunity costs.

The **opportunity cost** of a test refers to what you could have done instead, if you hadn't spent your resources running this test.

A common kind of discussion is whether achieving 5% more coverage of a certain kind is worth the opportunity cost (a different set of tests or reports will never be started if you spend your resources this way).

REVIEW (1)

- Designing for early testing
  - simple tests (e.g. function, domain, use-case, simple combinations)
- Designing for later testing
  - Complex combinations
  - Meaningful scenarios
  - Data-intense (or otherwise complex-to-test) risks
- Scenarios
  - Coherent story
  - Credible
  - Motivating
  - Complex
  - Easy to evaluate

REVIEW (2)

Good test design involves developing tests that

- Can help you satisfy your information objectives for this project (or this part of it)
- Address the things that you want to test in ways that can reveal the information that you want to find out about them
- Are achievable within your constraints
- Include the support materials (code, documentation, etc.) you will need for the level of reuse you consider appropriate
- Are optimized for the qualities (e.g. power) most important for your purposes

No one technique will fill all of your needs. Use many techniques, designing each test in a way that makes a given design problem seem easy and straightforward.

End of Lecture 4

Black Box Software Testing: Introduction to Test Design: Lecture 5: Domain Testing

Cem Kaner, J.D., Ph.D.
Professor of Software Engineering: Florida Tech

Rebecca L. Fiedler, M.B.A., Ph.D.
President: Kaner, Fiedler & Associates

These notes are partially based on research that was supported by NSF Grants EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers" and CCLI-0717613 "Adaptation & Implementation of an Activity-Based Online or Hybrid Course in Software Testing." 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.

OVERVIEW OF THE COURSE

	Technique	Context / Evaluation
1	Function testing & tours.	A taxonomy of test techniques.
2	Risk-based testing, failure mode analysis and quicktests	Testing strategy. Introducing the Heuristic Test Strategy Model.
3	Specification-based testing.	(… work on your assignment …)
4	Use cases and scenarios.	Comparatively evaluating techniques.
5	Domain testing: traditional and risk-based	When you enter data, any part of the program that uses that data is a risk. Are you designing for that?
6	Testing combinations of independent and interacting variables.	Combinatorial, scenario-based, risk-based and logical-implication analyses of multiple variables.

LECTURE 5 READINGS

Required reading

- (none for this lecture)

Recommended reading

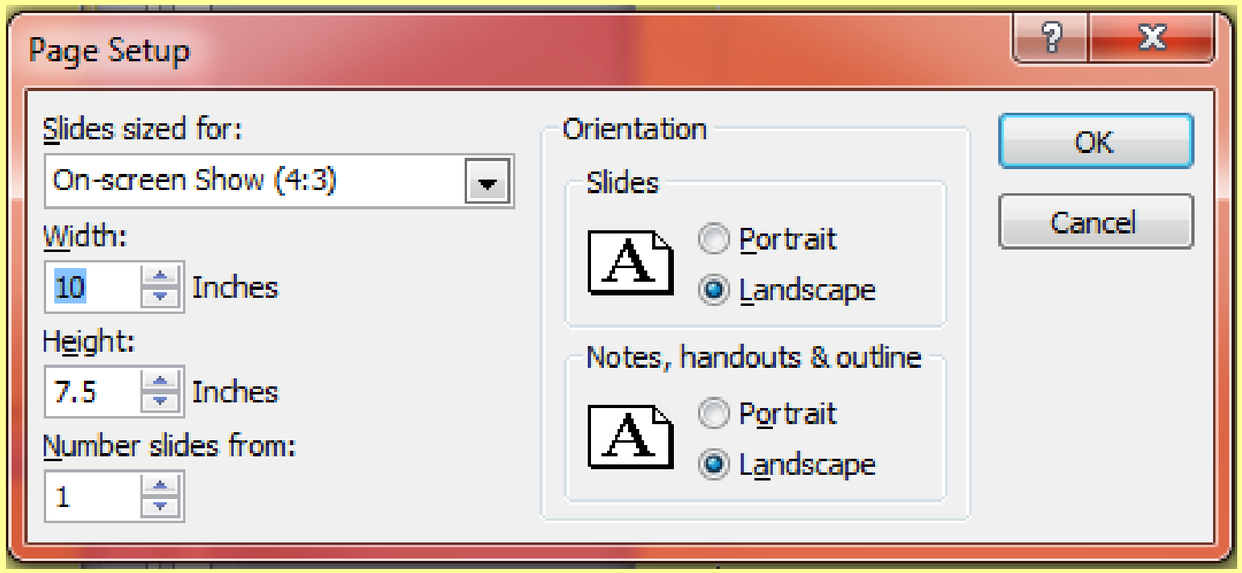
- Hamlet, D. & Taylor, R. (1990). Partition testing does not inspire confidence. IEEE Transactions on Software Engineering, 16(12), 1402-1411
- Kaner, C. (2004). Teaching domain testing: A status report. Paper presented at the Conference on Software Engineering Education & Training. http://www.kaner.com/pdfs/teaching_sw_testing.pdf
- Kaner, C., Hoffman, D., & Padmanabhan, S. (2012). Domain Testing: A Workbook. (To be published.)
- Myers, G. J. (1979). The Art of Software Testing. Wiley
- Padmanabhan, S. (2004). Domain Testing: Divide and Conquer. M.Sc. Thesis, Florida Institute of Technology. http://www.testingeducation.org/a/DTD&C.pdf
- http://www.wikipedia.org/wiki/Stratified_sampling

Your class might use other readings, but we had these in mind when creating this lecture.

Today's learning objectives

- Able to apply the traditional approach to straightforward cases.
- Aware of the underlying complexity of "equivalence" and "boundary." Understand the basis for the claim that partitioning and selecting boundaries require judgment, not just mechanical application of algorithms.
- Understand the difference(s) between primary and secondary dimensions.
- Understand the differences between input variables and result variables, and applicability of domain testing to both.
- Familiar with a conceptual structure for applying this analysis to a broad range of situations.

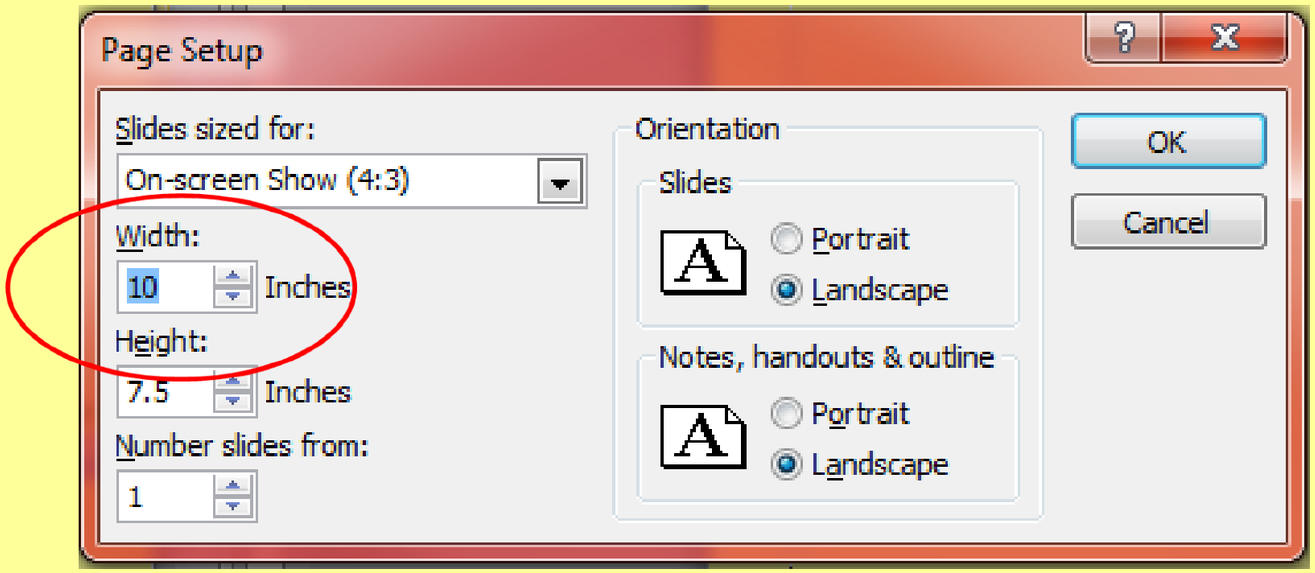
OPENING EXAMPLE



PowerPoint's **`Page Setup`** dialog lets you specify several aspects of the design of a slide.

Let's focus on one of them, **`Page Width`**.

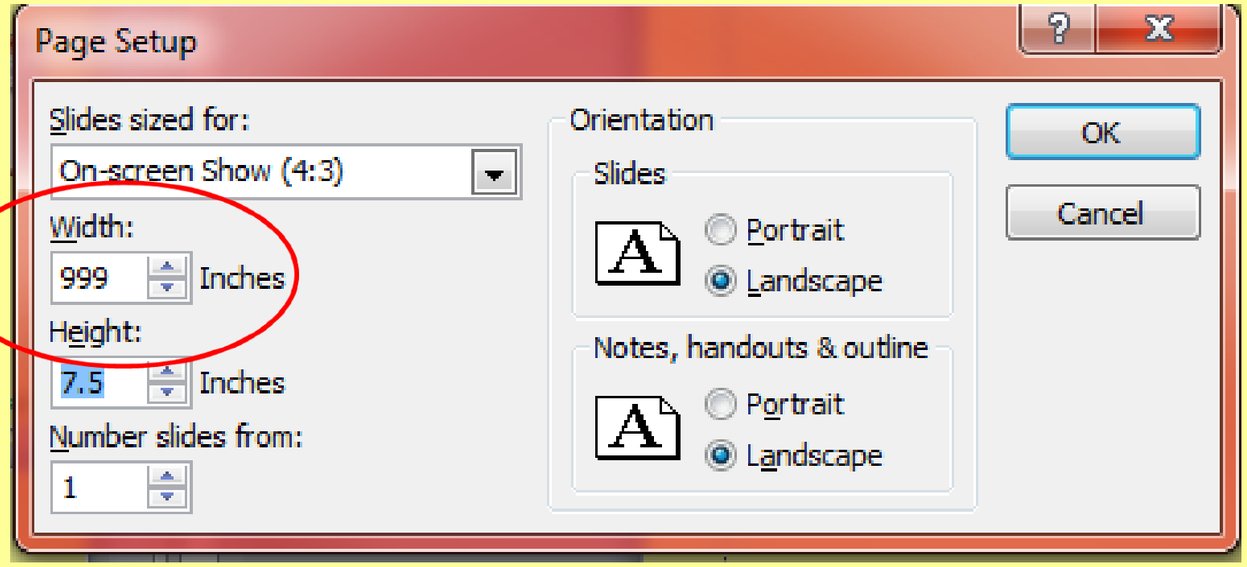
OPENING EXAMPLE



How wide is the widest page?

- I don't see an answer in Help or any documentation.
- ***Should I insist on a specification that tells me the range of this variable?***
- I'll try a big number and see what happens.

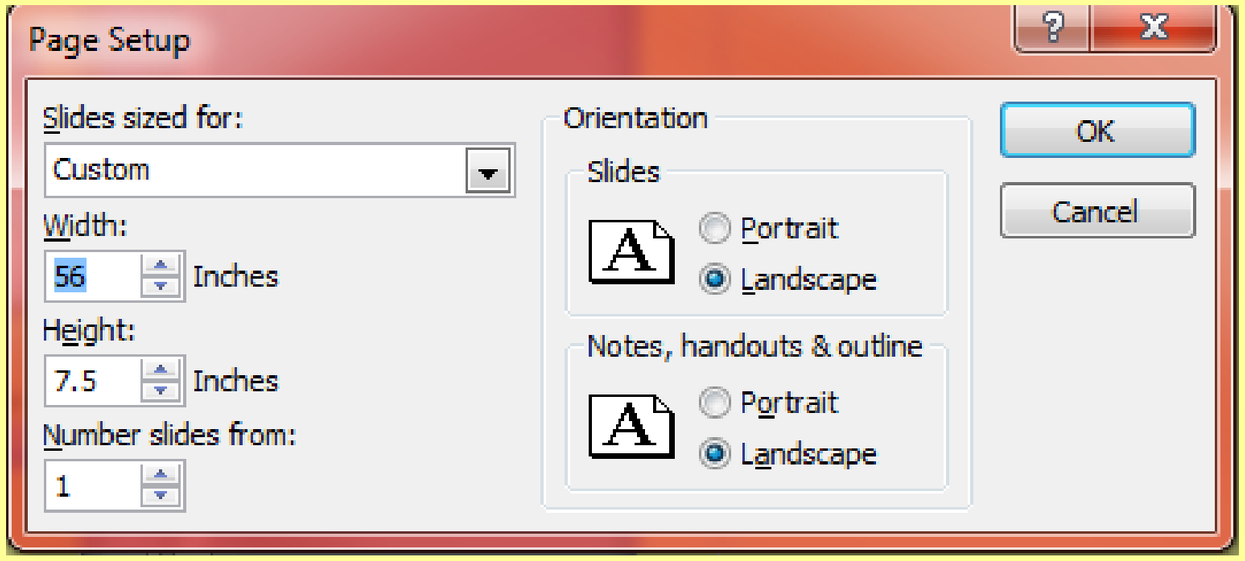
OPENING EXAMPLE



How wide is the widest page?

- I type in 999 inches

OPENING EXAMPLE



When I click OK, PowerPoint changes the 999 to 56 inches.

- I guess that's the limit.

OPENING EXAMPLE

So what happens if I change the width from 10 inches to 56?

A blank slide looks like this at 10

And like this at 56

OPENING EXAMPLE

As I try out the dialog, I see that it accepts two digits after the decimal point.

- It treats 10.12
  - the same as 10.123
  - but differently from 10.129
  - which it turns into 10.13

OPENING EXAMPLE

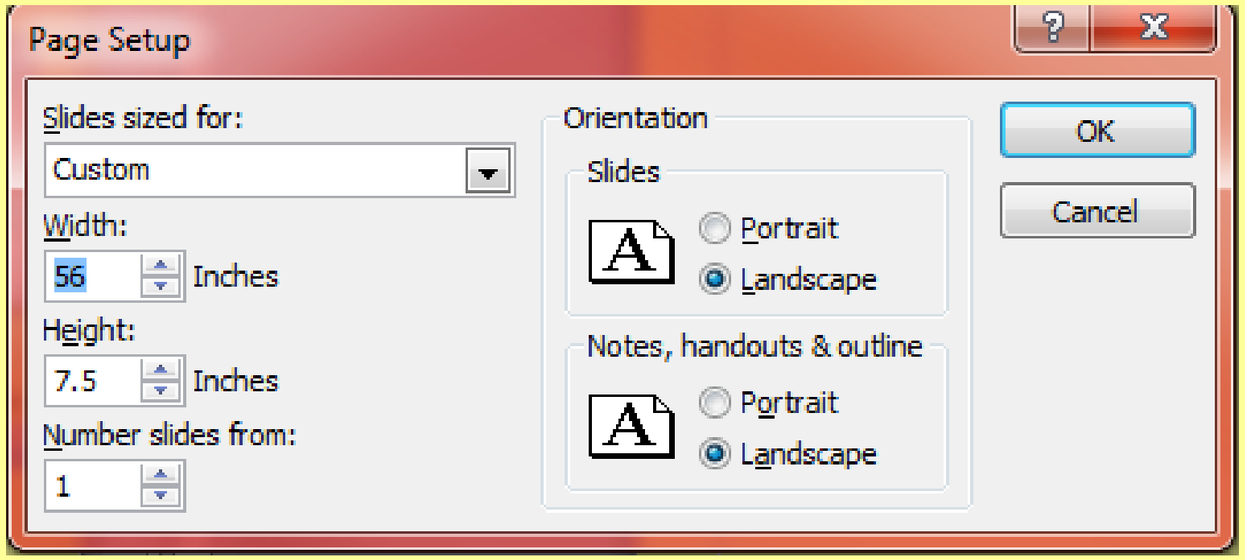
We've been inferring the design of the program from its behavior.

- Sometimes you have a specification.
- Often, all you have is
  - the behavior, and
  - your common sense, and
  - your skill as empirical researchers.

When you infer the design from behavior, pay careful attention to how the program uses that variable or feature. Are any details of the use inconsistent with the apparent design?

All of the oracle heuristics that we studied in Foundations are relevant to evaluating the "facts" that we discover about the program's design.

OPENING EXAMPLE



- The upper limit on width appears to be 56 inches.
- Try 0 for a lower limit:
  - PowerPoint changes it to 1
  - PowerPoint changes 0.9 to 1
- Therefore, the lower limit on width appears to be 1 inch.

OPENING EXAMPLE

What should I test?

- You could test:

  1.00, 1.01, 1.02, 1.03, …

  all the way through

  … 55.97, 55.98, 55.99, 56.00

- There are 5601 possible tests from 1 inch to 56 inches.
- What will you learn from testing 15.44 that you won't have already learned from 15.43?

Rather than running all 5601 tests, you need a sampling strategy.

The problem you have to solve

- You cannot afford to run every possible test.
- You need a method for choosing a few powerful tests that will represent the rest.

Domain testing (boundary and equivalence class analysis) is our field's most widely used way to address this problem.

DOMAIN DEFINITIONS

A domain is a set of values associated with a function.

Consider the variables X and Y and the function f, where

$$y = f(x)$$

- The **input domain** is the set of values over which the function is defined. This is the set of values of X.
  - We will call X the **input variable.**
- The **output domain** is the set of possible outputs of the function. This is the set of values of Y.
  - The output domain is also called the **range** of the function.
  - We will call Y the **result variable.**

Domain testing treats the program as a collection of functions that process input data in order to provide results.

Domain testing selects optimal values of the input domain or output domain for testing.

EQUIVALENCE

- Treat two tests as equivalent if:
  - they are so similar that
  - it seems pointless to test them both.
- Thus,
  - Two tests are equivalent if you expect the same results from each
- An equivalence class is a set of equivalent tests.

EQUIVALENCE

Treat

- individual values as
- equivalent to the nearest boundary value (1 and 56)
- if you believe that,
  - for any value X not on the boundary
  - if the program fails with X, it will also fail with one or both of the boundary cases.

We identified four substantially different definitions of equivalence in Kaner (2004) and Kaner, Padmanabhan & Hoffman (2012), but we'll stick with this subjective definition in this course.

Boundary cases

The dialog is designed to reject (replace) any value

- less than 1 or
- greater than 56.

Programmers often make a coding error at the boundary:

- accepting as valid a value that is barely too small or barely too large, or
- rejecting as invalid a value that is the smallest valid or the largest valid one.

In testing a range like 1 to 56, testers often test just at the boundaries (1 and 56), treating the interior values as equivalent to these.

CHECK THE INVALID VALUES

Just as you should test the "valid" boundaries, you should test the invalid ones:

- 0.99 inches
- 56.01 inches.

If the program is going to erroneously treat any invalid input as if it were valid, it will make that mistake with boundary cases because they are the closest values to the valid ones.

The classic boundary / equivalence class table

Variable	Valid case equivalence class	Invalid case equivalence class	Boundaries & special cases	Notes
Page width		< 1	0.99	Fixed-point variable rounded to 2 digits after decimal point.
	1 to 56 inches		1.0	
			56	
		> 56	56.01	

Don't do this

- Some people pack all the sets and all the tests into one row of the table.
- This is easy to understand while you're creating it, but harder to read later.
- It is too easy to not notice important tests or to get confused about the reason some test was included.

Variable	Valid case equivalence class	Invalid case equivalence class	Boundaries & special cases	Notes
Page width	1 to 56 inches	< 1, > 56	0.99, 1, 56, 56.01	Fixed-point variable rounded to 2 digits after decimal point.

Do this

- Separate the tests for the different equivalence classes
- One test per line
- Make notes to explain any facts about the variable or reasoning about the tests that are not obvious

Variable	Valid case equivalence class	Invalid case equivalence class	Boundaries & special cases	Notes
Page width		< 1	0.99	Fixed-point variable rounded to 2 digits after decimal point.
	1 to 56 inches		1.0	
			56	
		> 56	56.01	

CONSIDERING THE CONSEQUENCES

It's not enough to just check whether the program accepts valid inputs and rejects invalid inputs.

- In unit testing, checking the input filters is appropriate
- In system testing, checking only the input filters, without considering the consequences of the input values, is a hallmark of amateurish testing.

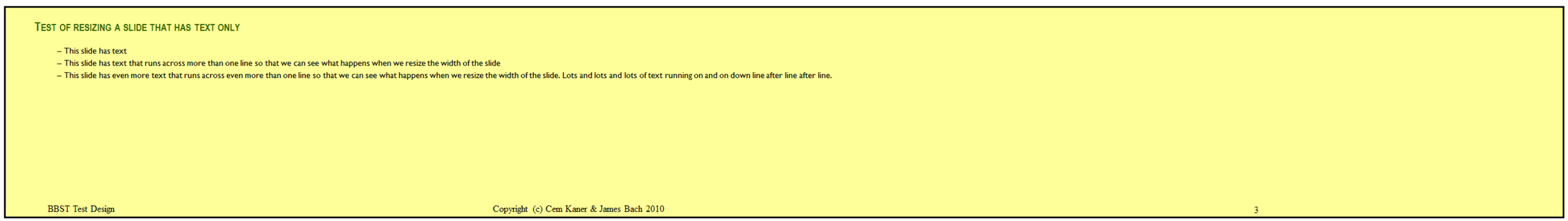
Test of resizing a slide that has text only

- This slide has text
- This slide has text that runs across more than one line so that you can see what happens when you resize the width of the slide
- This slide has even more text that runs across even more than one line so that you can see what happens when you resize the width of the slide. Lots and lots and lots of text running on and on down line after line after line.

The next cluster of slides illustrates some tests of consequences of changing page width.

Test of resizing a slide that has text only

- Test with 56 inches wide and 7.5 inches tall.
- The slide stretches to become very wide
- The text stays the same size (e.g. 24 points) but is no longer wrapped because it fits on one line
- The slide stretches so much that when we paste it here, the text is unreadably small. Let's rescale to a narrower still-wide slide.



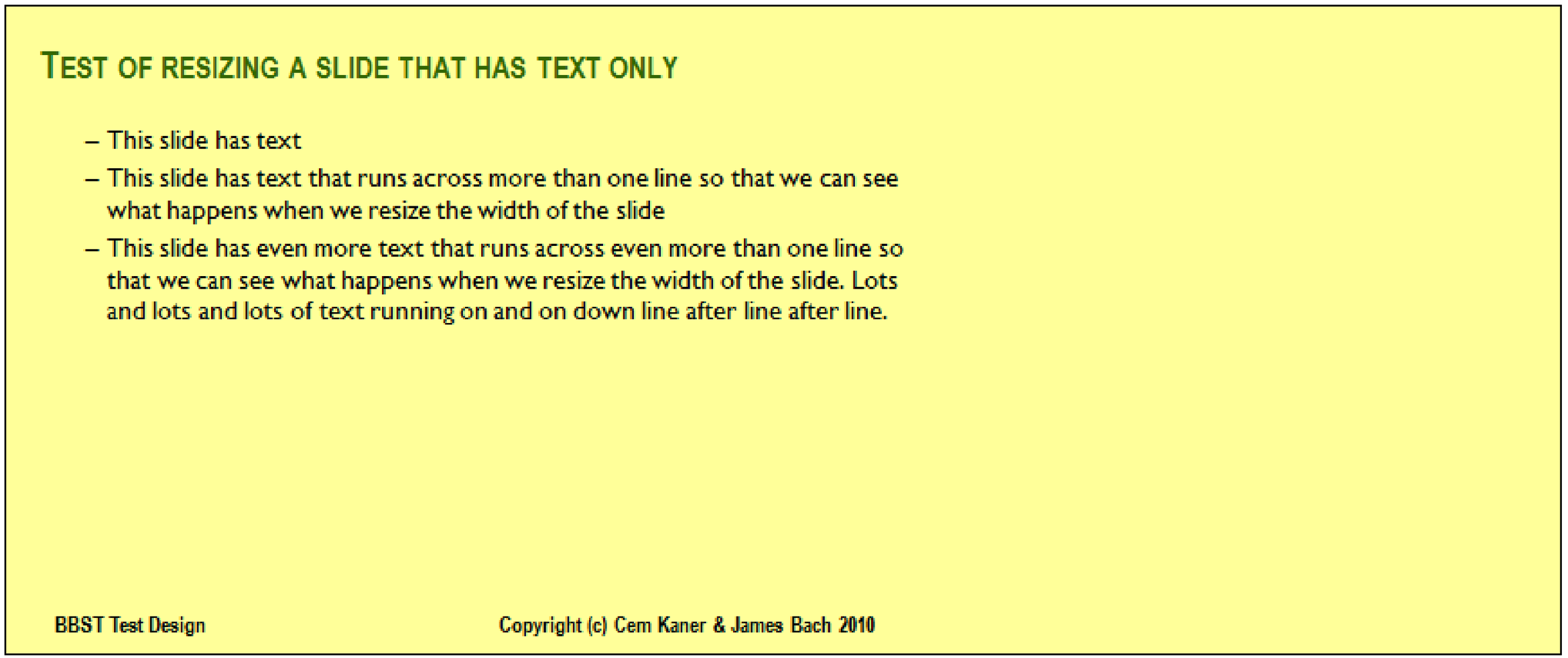
Test of resizing a slide that has text only

- This slide has text
- This slide has text that runs across more than one line so that you can see what happens when you resize the width of the slide
- This slide has even more text that runs across even more than one line so that you can see what happens when you resize the width of the slide. Lots and lots and lots of text running on and on down line after line after line.

This is a copy of the original again...

Test of resizing a slide that has text only

- 18 inches wide and 7.5 inches tall.



Test resizing a slide with a table

Variable	Valid case equivalence class	Invalid case equivalence class	Boundaries & special cases	Notes
Page width	1 to 56 inches		1.0	Fixed-point variable rounded to 2 digits after decimal point.
			56	
		< 1	0.99	
		> 56	56.01	

TEST RESIZING A SLIDE WITH A TABLE

- 18 inches wide and 7.5 inches tall.
- The columns are wider. The text is the same

TEST RESIZING A SLIDE WITHA TABLE

Variable	Valid case equivalence class	Invalid case equivalence class	Boundaries & special cases	Notes
Page width	1 to 56 inches		1.0	Fixed-point variable rounded to 2 digits after decimal point.
			56	
		< 1	0.99	
		> 56	56.01	

BBST Test Design Copyright (c) Cem Kaner & James Bach 2010

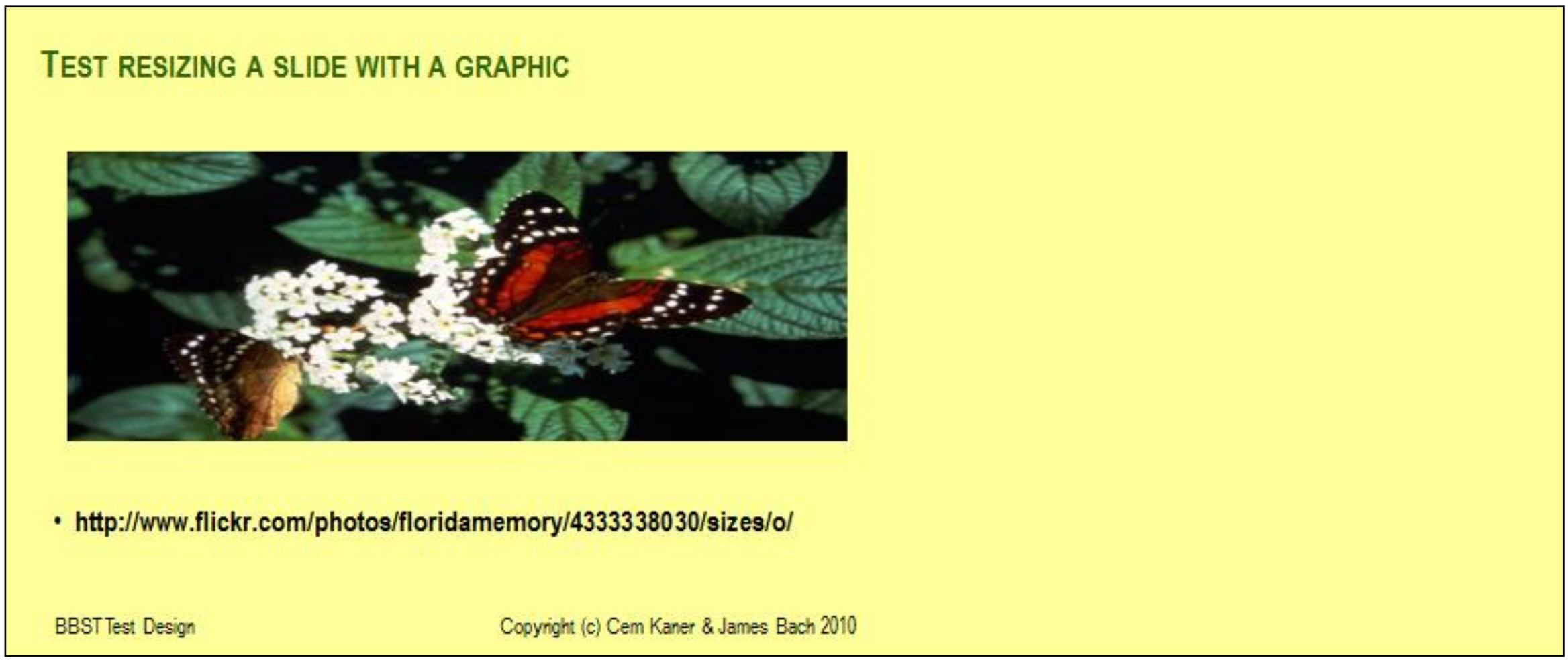
Test resizing a slide with a graphic

http://www.flickr.com/photos/floridamemory/4333338030/sizes/o/

Test resizing a slide with a graphic

18 inches wide * 7.5 inches tall

The text stays the same size. The graphic is stretched.



Test resizing a slide with a graphic

25 inches wide * 7.5 inches tall

The text stays the same size. The graphic is even more obviously stretched.

Test resizing a slide with a graphic

- http://www.flickr.com/photos/floridamemory/4333338030/sizes/o/

Test resizing a slide with an imported table

Variable	Valid Case Equivalence Classes	Invalid Case Equivalence Classes	Boundaries and Special Cases	Notes
First number	-99 to 99	> 99 < -99	99, 100 -99, -100	
Second number	-99 to 99	> 99 < -99	99, 100 -99, -100	

TEST RESIZING A SLIDE WITH AN IMPORTED TABLE

- 18 inches wide and 7.5 inches tall.
- Everything inside the table is resized.

TEST RESIZING A SLIDE WITH AN IMPORTED TABLE

Variable	Valid Case Equivalence Classes	Invalid Case Equivalence Classes	Boundaries and Special Cases	Notes
First number	-99 to 99	> 99 < -99	99, 100 -99, -100	
Second number	-99 to 99	> 99 < -99	99, 100 -99, -100	

BBST Test Design

Test resizing a slide with an imported table

- 25 inches wide and 7.5 inches tall.
- Shows the resizing (and distortion of the text) even more clearly.

Test resizing a slide with an imported table

Variable	Valid Case Equivalence Classes	Invalid Case Equivalence Classes	Boundaries and Special Cases	Notes
First number	-99 to 99	> 99 < -99	99, 100 -99, -100	
Second number	-99 to 99	> 99 < -99	99, 100 -99, -100	

TESTS OF RESIZING

- Sowmya Padmanabhan (2004) used slide resizing as part of her thesis research on how people learn domain testing
- At that time,
  - text was also distorted
  - resizing the page multiple times often
    - corrupted the graphics
    - after more resizes, crashed.
- Despite extensive training, not one student found this bug.

- **The input field <u>filter</u> was fine.**
- **Handling of the actual data that was input was broken.**

TESTING FOR CONSEQUENCES

- People don't buy a program so that they can enter data into fields like page width
- They buy it to create things (analyses or graphics or ....)
  - They enter the data
  - for the program to use the data
  - to achieve the result.
- If you don't test the result (the effect of the data you enter)
  - Your tests are missing the point.
  - Their value is minimal.

You should be especially interested in the consequences when the program allows you to enter an invalid value.

Summary to this point

We have studied several parts of the domain testing process:

- Identify the variable of interest.
- Determine the type of the variable and the values it can take.
- Determine how the program uses this variable.
- Partition the variable into valid and invalid equivalence classes.
- Test with boundary values.
- Test for consequences of the data entered, not just the input filter.
- Describe the tests in a classical boundary / equivalence class table.

See the slides at the end of the lecture for a more complete schema for domain testing.

Definitions

Input domain: the set of possible values that you can input to the variable.

Output domain: the set of possible values of an output variable (such as the actual displayed width of the slide).

Equivalent values: two or more values of a domain that you expect to yield equivalent (pass/fail) test results.

Equivalence class: a subset of a domain that has equivalent elements.

Partition: separation of a domain into non-overlapping equivalence classes.

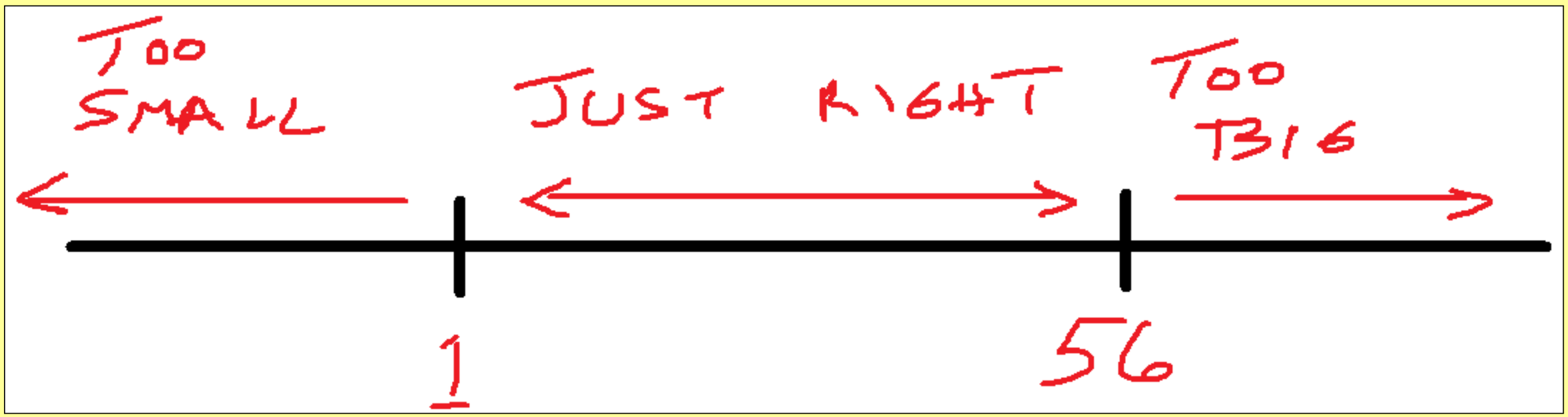
Input filter: code that blocks input of invalid values.

In domain testing, we partition a domain into sub-domains (equivalence classes) and then test using best representatives (e.g. boundary values) from each sub-domain.

Primary dimension of a variable

The **`Page Width`** variable is looking for values that run from 1 to 56.

- A variable doesn't only vary on a single dimension. However, some of these are incidental, having more to do with the implementation than with the purpose of the variable.
- You can usually determine the primary dimension by asking what you're trying to control or to learn from the variable.



SECONDARY DIMENSIONS

The non-primary dimensions on which a variable can vary. Examples:

- Number of digits
  - 0 (empty field)
  - 1 to 4 (1 to 56.00)
  - 5 or more
  - The most interesting test might use thousands of digits
- The character set (ASCII codes)
  - 0 to 47 ("/" is 47) (non-digits)
  - 48 to 57 ("0" to "9") (digits)
  - 58 to 127 (":" is 58) (non-digits)

Myers' (1979) triangle program is one of the classic examples of domain testing. His analysis (and many others (e.g. Binder, 2000, p. 5) provide many tests along secondary dimensions (e.g. testing non-numbers) but they don't distinguish between primary and secondary dimensions. This has caused much confusion.

SECONDARY DIMENSIONS

Examples

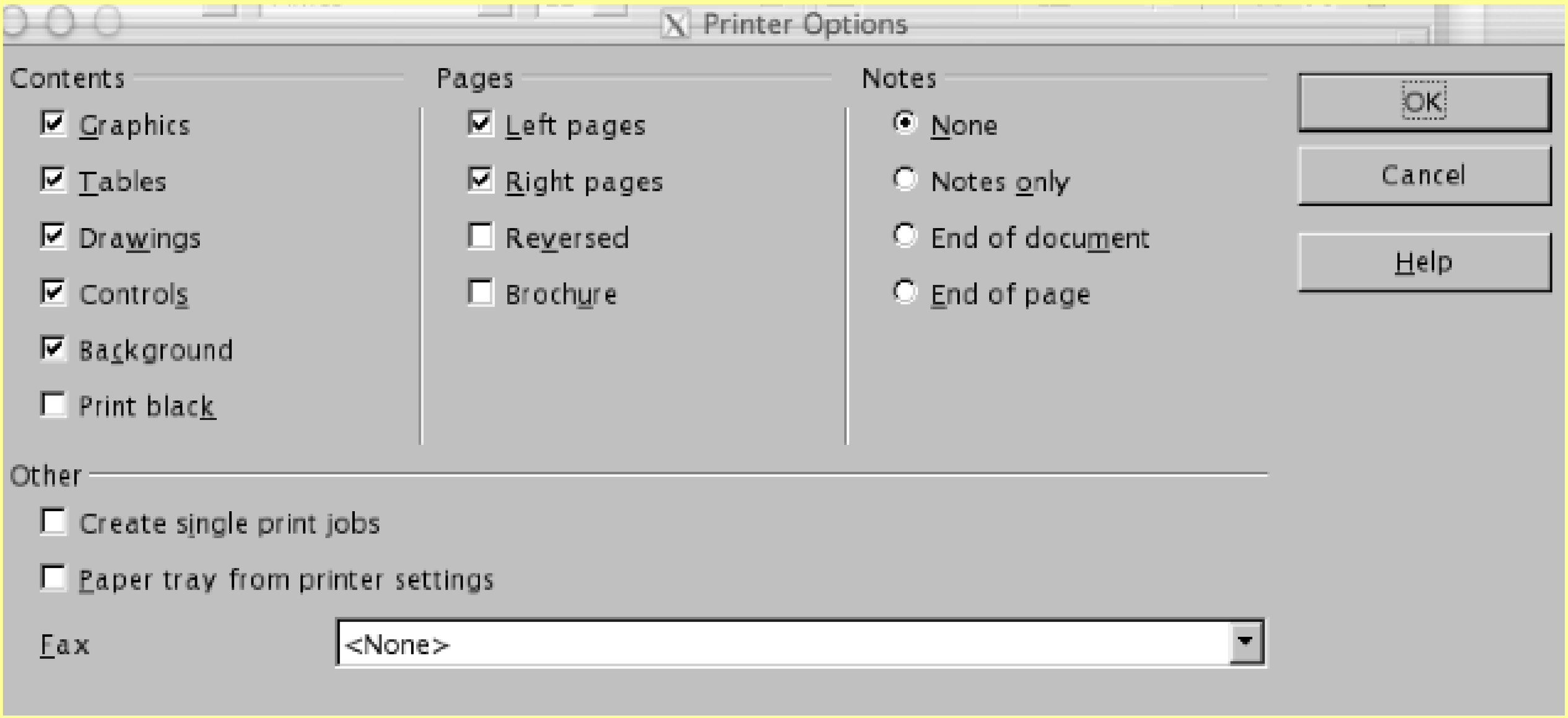
- Number of digits
- Character codes (ASCII)
- Leading spaces
  - none (this is the typical case)
  - 1 (not unusual)
  - >1 (how many can you have?)
- Spaces between digits
  - 0 (typical)
  - >0 (OpenOffice ignores "invalid" characters inside a number string)

Secondary dimensions on the classical table

Variable	Valid case equivalence class	Invalid case equivalence class	Boundaries & special cases	Notes
Page width	1 to 56 inches		1.0	Fixed-point variable rounded to 2 digits after decimal point.
			56	
		< 1	0.99	
		> 56	56.01	
	1 to 4 characters		1	
			55.99	
		0 (no characters)	Delete the value in the field	
		> 5	55.999	Easy to pass
			55.9999…	1000 digits

Some primary dimensions are not appropriate for domain testing

Domain analysis is pointless with binary variables because there are no "equivalent" values that you can skip.



Secondary dimensions on the classical table

Variable	Valid case equivalence class	Invalid case equivalence class	Boundaries & special cases	Notes
Page width	1 to 56 inches		1.0	Fixed-point variable rounded to 2 digits after decimal point.
			56	
		< 1	0.99	
		> 56	56.01	
	1 to 4 characters		1	
			55.99	
		0 (no characters)	Delete the value in the field	
		> 5	55.999	Easy to pass
			55.9999…	1000 digits

Choosing boundaries

- If you can order the values that a variable can take, from smallest to largest:
  - The upper boundary of an equivalence class is the largest value in the set. Call this boundary value UB.
  - Let Δ (delta) be the smallest possible difference between two values
    - Between integers, Δ = 1
    - Between fixed-point with 5 significant digits after the decimal, Δ = 0.00001
- The next boundary of interest is UB + Δ

Page width example:

UB + Δ = 56.01

UB = 56

LB = 10

LB - Δ = 9.99

Multiple valid classes

Think of course grades:

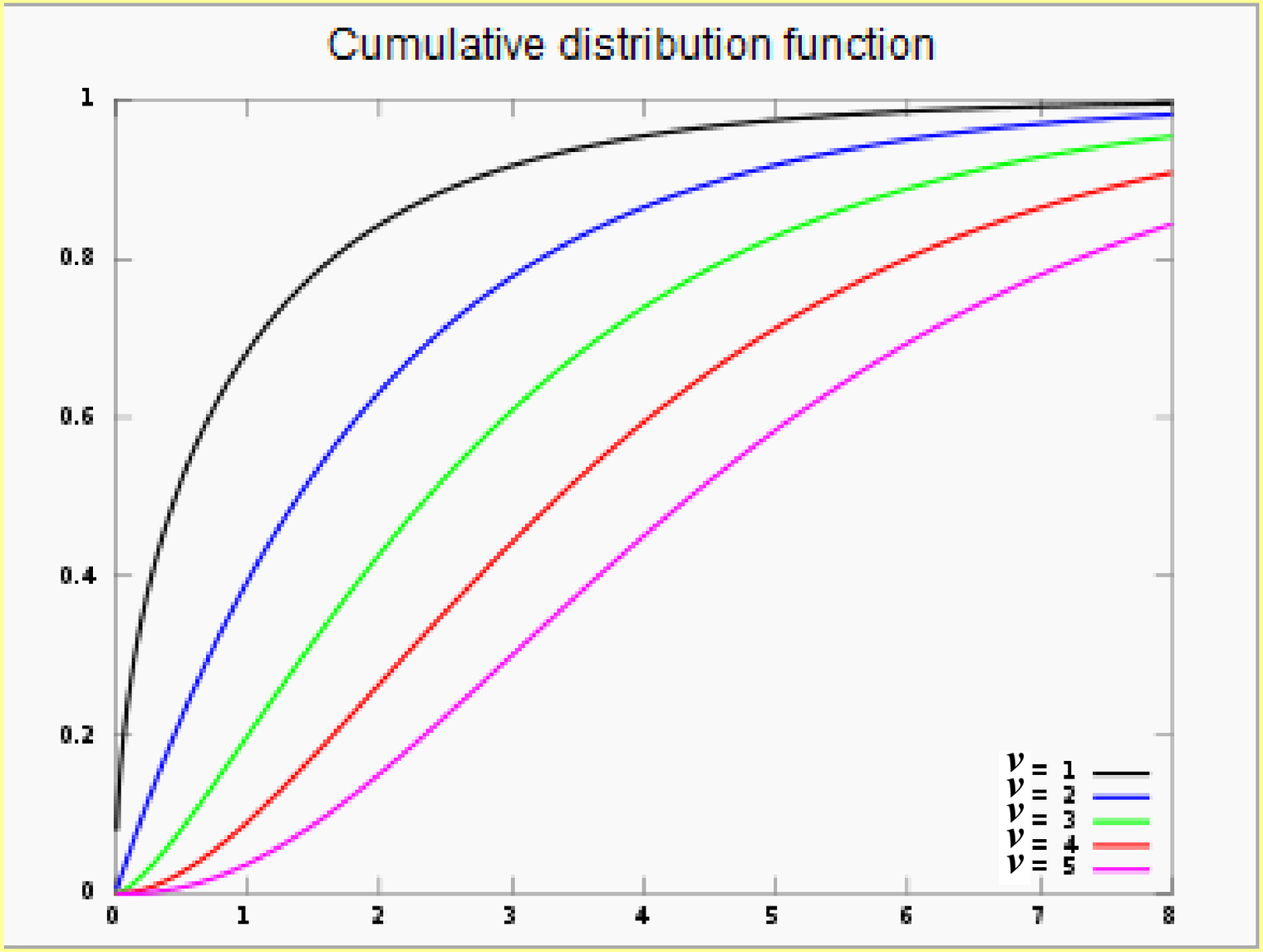
- A (90 to 100)
- B (80 to 89)
- C (70 to 79)
- D (60 to 69)
- F (0 to 59)

How should you show these in the table?

MULTIPLE VALID CLASSES

Variable	Valid case equivalence class	Invalid case equivalence class	Boundaries & special cases	Notes
Grade		> 100	101	Δ is 1
	90 - 100		100	A
			90	
	80 - 89		89	B
			80	
	70 - 79		79	C
			70	
	60 – 69		69	D
			60	
	50 - 59		59	F
			0	
		< 0	-1	

HIDDEN BOUNDARIES



http://en.wikipedia.org/wiki/Chi-square_distribution

The cumulative chi-square probability distribution function

$$Q(\chi^2|\nu) = 1 - P(\chi^2|\nu) \qquad (0 \leq \chi^2 < \infty)$$

$$= \left[2^{\nu/2}\,\Gamma\left(\frac{\nu}{2}\right)\right]^{-1} \int_{\chi^2}^{\infty} (t)^{\frac{\nu}{2}-1} e^{-\frac{t}{2}}\, dt$$

Abramowitz & Stegun (1964),

The shape parameter of this function is ν ("nu").

HIDDEN BOUNDARIES

- A program might use 3 or more different formulas to calculate Chi-Square probability values, depending on v, the number of degrees of freedom.
- Unless you know this internal implementation detail, your testing will probably treat $v = 31$ in the same equivalence class as $v = 101$, even though the approximation formulas are entirely different.

Approximations to the Chi-Square Distribution for Large ν

26.4.13

Approximation		*Condition*
$Q(\chi^2\|\nu) \approx Q(x_1)$,	$x_1 = \sqrt{2\chi^2} - \sqrt{2\nu - 1}$	$(\nu > 100)$

26.4.14

$$Q(\chi^2|\nu) \approx Q(x_2), \qquad x_2 = \frac{(\chi^2/\nu)^{1/3} - \left(1 - \frac{2}{9\nu}\right)}{\sqrt{2/9\nu}} \qquad (\nu > 30)$$

This is an example of the subjectivity of testers' classifications into equivalence classes.

Abramowitz & Stegun (1964), Handbook of Mathematical Functions, p. 941, http://people.math.sfu.ca/~cbm/aands/frameindex.htm

EQUIVALENCE IS RISK-BASED

Consider the **`Page Width`** example again.

1) All values greater than 56 are equivalent **relative to the risk** that the program's error handling fails with values > 56.
2) In addition, the program might fail when it incorrectly accepts 56.01 as valid (The programmer wrote the inequality wrong: Statement d instead of Statement a or b).

56.01 belongs to two equivalence classes. It can trigger a failure in two ways (1 & 2). The other members of the class only have the one way (1).

The code might CORRECTLY say:

a. Accept all X ≤ 56

Or b. Accept all X < 56.01

Or it might INCORRECTLY say:

c. Accept all X < 56

Or d. Accept all X ≤ 56.01

EQUIVALENCE IS RISK-BASED

- Consider the risk of an input overflow.
  - How many digits can the program cope with?
    - Will it fail with an entry of 999? 9999?
    - What about 999… (255 characters)?
  - Suppose the system is designed to truncate any input string > 5 characters.
    - Relative to the input overflow risk:
      - » Entries with 1 to 5 digits are equivalent.
      - » 6 chars is the smallest out-of-bounds case.
      - » 56.01 is equivalent to 99,999 but not 100,000.

EQUIVALENCE IS RISK-BASED

Suppose { x_1, x_2, x_3, …, x_n } is a set of equivalent values.

- These are equivalent with respect to some risk
- But they might not be equivalent with respect to some other risks
- For example, as you've seen:

  {56.01, …, 99999, 100,000}

  are equivalent with respect to some risks
  - (they're all bigger than 56),

  But not with respect to others
  - 56.01 is in its own boundary-risk class
  - 100,000 is in the greater-than-6-digits class, but 56.01 is not.

Best representative

- A **best representative** of an equivalence class is the test within that class most likely to make the program fail.
- Often, the best representative is:
  - **at least as likely to trigger a failure as any other member of the set (when you consider the risk they are equivalent against)**
    - 56.01 is at least as likely to trigger a bigger-than-56 failure as any other value bigger than 56
  - **more likely to trigger a failure than the other members (relative to some other risk)**
    - 56.01 can trigger a boundary failure. None of the other members of the bigger-than-56 class can do that.

Boundaries (extreme values) are typical best representatives.
A set can have more than one "best representative": think of a set that has more than one boundary

Non-ordered variables

Imagine testing a program's compatibility with printers.

- There are thousands of different printers, so you need a sampling strategy
- Most printers are "compatible" with some other printer(s), so you can group printers into equivalence classes.
- But you can't put printers into an order, so "boundary values" don't exist.
- Best representatives of a compatibility set will differ from the others in terms of vulnerability to some other risk (e.g. memory management)

Kaner, Falk & Nguyen (1993) presented this analysis in detail.

Non-ordered variables

More examples:

- don't fit the traditional mold for equivalence classes
- so many values that you must sample from them.
- What are their boundary cases?

- Membership in a common group
  - employees vs. non-employees
  - full-time vs. part-time vs. contract
- Equivalent output events
  - perhaps any report will do to answer a simple question like: Will the program print reports?
- Equivalent environments
  - different languages, same O/S

Sometimes, a set has no best representative. If there is no reason to choose one member of a set over another, there is no best representative (or all of them are).

In sum: equivalence classes and representative values

- Two tests belong to the same **equivalence class** if you expect the same result (pass / fail) of each. Testing multiple members of the same equivalence class is, by definition, redundant testing.
- In an ordered set, **boundaries** mark the point or zone of transition from one equivalence class to another. The program is more likely to fail at a boundary, so these are the best members of (simple, numeric) equivalence classes to use.
- More generally, you look to subdivide a space of possible tests into relatively few classes and to run a few cases of each. You'd like to pick the most powerful tests from each class. We call those most powerful tests the **best representatives** of the class.
- Xref: stratified sampling: http://www.wikipedia.org/wiki/Stratified_sampling

Summary of our process (so far)

- Identify the variable of interest.
- Identify its primary dimension.
- Determine the type of the variable (along the primary dimension) and the values it can take.
- Determine how the program uses this variable.
- Determine whether you can order the variable's values (from smallest to largest).
- Partition the variable's domain into equivalence classes.
- Test with best representatives.
- Test for consequences of the data entered, not just the input filter.
- Describe the tests in a classical boundary / equivalence class table.
- Identify secondary dimensions. Analyze them in the traditional way.

In domain testing, we partition a domain into sub-domains (equivalence classes) and then test using best representatives (e.g. boundary values from each sub-domain.

The Myers Example

"Before beginning this book, it is strongly recommended that you take the following short test. The problem is the testing of the following program:

> The program reads three integer values from a card. The three values are interpreted as representing the three sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral.

On a sheet of paper, write a set of test cases (i.e. specific sets of data) that you feel would adequately test this program."

Glen Myers (1979).

The Art of Software Testing.

Page 1

COMMON TEST IDEAS FOR PAGE WIDTH (FLOATING POINT FIELDS)

Page too small (< 1)	Too few characters	Leading zero	Unicode chars not in ASCII
Lower bound (1)	Max number of digits	Many leading zeros	No decimal point
Upper bound (56)	Max number of digits, plus a decimal point	Leading "+" sign	Two decimal points
Page too wide (> 56)	Max digits plus spaces	Many leading "+" signs	Commas (thousands separators)
0	Too many digits	Mix leading "+" and spaces	Commas in inappropriate places
Negative number	Way too many digits	Non-digits (such as "/" and ":")	Expressions
Far below the lower bound (e.g. -999…)	Whitespace only (spaces or tabs)	Uppercase letters	Scientific notation
Far above the upper bound	Leading space	Lowercase letters	Scientific notation with invalid values
Empty cell	Many leading spaces	Upper ASCII chars	Scientific notation: out-of-bounds values

A NEW TABLE: RISK / EQUIVALENCE

Variable	Risk (potential failure)	Class that should not trigger this failure	Class that might trigger this failure	Test cases (best representatives)	Notes
Page width	impossibly small page	≥ 1.00		1.00	
			< 1	0.99	
				0	
				-1	
	distorted graphics	don't resize the page		don't resize	
		resize a page that has no graphics		resize blank slide	
			Place graphics. Use different formats. Resize to different height/width ratio	1.00	stretch only width. Don't stretch width and height proportionally
				56.00	

Risk / Equivalence Analysis

Focus on the individual variable

- or on a small group of related variables
- (because that's what you do in domain testing)

Identify "all" the ways the variable could be involved in a failure

- For each risk, create equivalence classes
  - One set shouldn't trigger this failure (in the old jargon, "valid cases")
  - The others should have the potential to trigger the failure
  - The best representative in each class is the one most likely to trigger the failure.

Adding expected results to the tables.

- Some people prefer to add an **Expected Results** column to their domain testing tables
- You can add this column to either table, the classical one or the risk/equivalence one.
- What value is the expected value?
  - The one the program SHOULD give if it is working correctly
  - Not the one you hope to see if it fails.
  - Describe hoped-for failures in your **Notes** column.

Sometimes you'll run tests without knowing your expected results.

If you don't know the answer, try it and find out.

COMPARING THE TABLES

The Classical Table

The classical table excels at making boundary tests obvious, so that with a minimum of training, people can create the table or read and understand it.

The Risk / Equivalence Table

The risk-oriented table is a little more complex to work with when you are dealing with simple variables. We often prefer the classical table for simple, academic examples.

The weakness of the very simple examples is that they are divorced from real-life software. You analyze a variable, but you don't know why a program needs it, what the program will do with it, what other variables will be used in conjunction with it. As soon as you know the real-life information, many risks (should) become apparent, risks that you can study by testing different values of this variable. The risk-oriented table helps you organize that testing. Any time you are thinking beyond the basic "too big / too small" tests, this style of table might be more helpful than the classical one.

In Practice

I often create the classical table early in testing, shifting to the risk / equivalence table as I learn more about software under test.

RESULT VARIABLES

Suppose:

- I, J, and K are unsigned integers.
- K = f (I, J) = I * J
  - Input domain: {(I,J)}
  - Output domain {K}

K is a result variable.

- You can enter values into I and J.
- You cannot enter values into K.
- The program calculates the value of K.

RESULT VARIABLES

K = f (I, J) = I * J

Do a domain analysis on K.

- This is a reasonable requirement.
- It's like testing:
  - the balance (how much money you have) in your checking account
  - the amount on your paycheck (hours * rate of pay – deductions)

If you WERE testing I or J, you should also test K because the value of K is a consequence of the values you enter into I or J.

THE ANALYSIS (RESULT VARIABLE)

This table shows what values of K you want to test.

Now you have to figure out what values of I and J to use in order to generate those values of K.

Variable	Valid case equivalence class	Invalid case equivalence class	Boundaries & special cases	Notes
K	0 to MaxInt		0	Unsigned (all values positive)
			MaxInt	
		< 0	Can’t do that	I, J can't be negative
		> MaxInt	MaxInt + 1	
			MaxInt * MaxInt	

THE ANALYSIS (RESULT VARIABLE)

Consider K = 0. Several (I,J) pairs will yield this value of K.

An (I,J) pair includes a value of I and a value of J. For example,

(1, 2) means I = 1 and J =2

The full set can be described like this:

{ (I, J) | I * J = 0 }

This is read as “The set of all pairs of I and J such that I times J equals 0”.

THE ANALYSIS (RESULT VARIABLE)

Continuing the analysis:

{ (I, J) | I * J = 0 }

= { (I, J) | I = 0 or J = 0 }

This is an equivalence set on the (I, J)'s.
The set includes

- (0, 0),
- (1, 0),
- (MaxInt, 0)
- (0, 1)
- (0, MaxInt)
- intermediate values, like (0, 2000).

THE ANALYSIS (RESULT VARIABLE)

Variable	Valid case equivalence class	Invalid case equivalence class	Boundaries & special cases	I	J	Notes
K	0 to MaxInt		0	0	0	{ (I, J) | I = 0 or J = 0 }
				0	MaxInt	
				MaxInt	0	
			MaxInt			
		< 0	can't do that			
		> MaxInt	MaxInt + 1			
			MaxInt * MaxInt			

THE ANALYSIS (RESULT VARIABLE)

Continuing the analysis:

{ (I, J) | I * J = MaxInt }

For example, if MaxInt is 2^{16}-1, then this equivalence set on the (I, J)'s includes

- (1, MaxInt) and (MaxInt,1)
- (3, 21845) and (21845, 3)
- (5, 13107) and (13107, 5)
- (15, 4369) and (4369, 15)
- (17, 3855) and (3855, 17)
- (51, 1285) and (1285, 51)
- (85, 771) and (771, 85) and
- (255, 257) and (257, 255)

For other values of MaxInt, the set will be different.

THE ANALYSIS (RESULT VARIABLE)

Variable	Valid case equivalence class	Invalid case equivalence class	Boundaries & special cases	I	J	Notes
K	0 to MaxInt		0	0	0	{ (I, J) | I = 0 or J = 0 }
				0	MaxInt	
				MaxInt	0	
			MaxInt	1	MaxInt	{ (I, J) | I * J = MaxInt }
				MaxInt	1	
		< 0	can't do that			
		> MaxInt	MaxInt + 1			
			MaxInt * MaxInt			

THE ANALYSIS (RESULT VARIABLE)

Variable	Valid case equivalence class	Invalid case equivalence class	Boundaries & special cases	I	J	Notes
K	0 to MaxInt		0	0	0	{ (I, J) | I = 0 or J = 0 }
				0	MaxInt	
				MaxInt	0	
			MaxInt	1	MaxInt	{ (I, J) | I * J = MaxInt }
				MaxInt	1	
		< 0	can't do that			
		> MaxInt	MaxInt + 1	2^(N/2)	2^(N/2)	where MaxInt = 2^N - 1
			MaxInt * MaxInt	MaxInt	MaxInt	

Result variables: Generalizing the notation

$(y_1, y_2, \ldots, y_m) = f(x_1, x_2, \ldots, x_n)$

- $X = (x_1, x_2, \ldots, x_n)$ is the input variable
- $\{ (x_1, x_2, \ldots, x_n) \}$ is the input domain
- $Y = (y_1, y_2, \ldots, y_m)$ is the result variable
- $\{(y_1, y_2, \ldots, y_m) \}$ is the output domain

$$Y = f(X)$$

is the same as

$$(y_1, y_2, \ldots, y_m) = f(x_1, x_2, \ldots, x_n)$$

Reminder:

(I, J) is a 2-tuple, and

$(x_1, x_2, \ldots, x_n)$ is an n-tuple .

If you've forgotten this, look back at your course notes from Foundations.

Result variables: a 4-step summary

When Y = F(X),

To test Y

1. Figure out what values of Y you want to test
2. Figure out what values of X will generate that value of Y
3. For a given Y, there will be an equivalence set of X values. Identify the set
4. Pick one or more X's from the set.

This is just the first part of the design of the test.

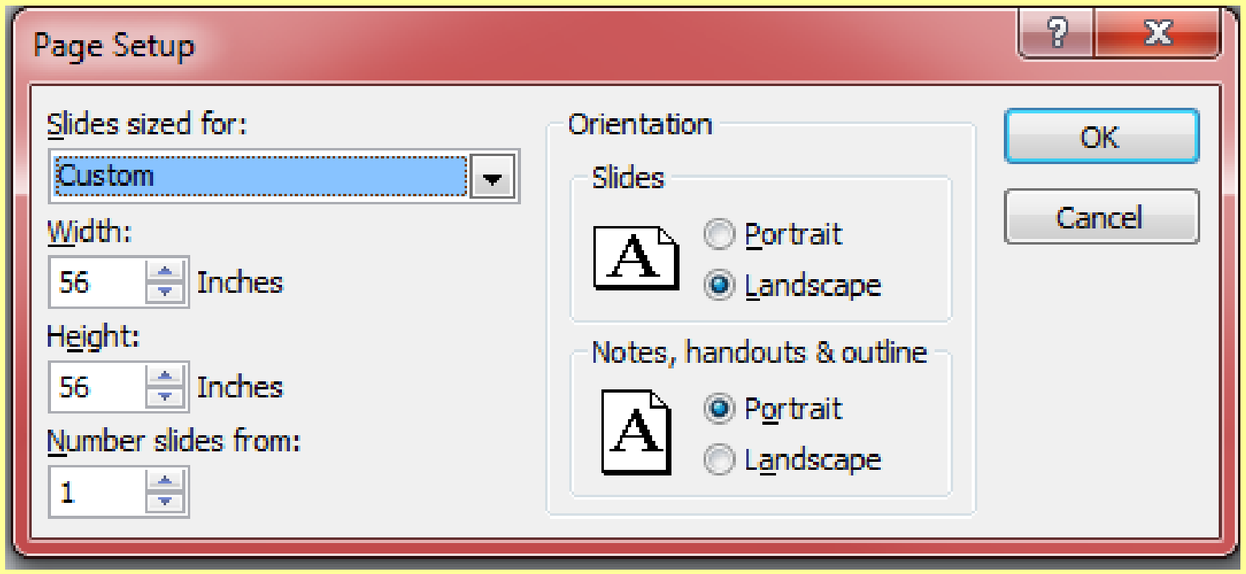
- **What are the consequences?**
- **What are the oracles (for the result, and for each consequence)?**

Looking ahead at multi-dimensional variables

Independent Components

Page setup has five independent variables:

- Page width (1 to 56)
- Page Height (1 to 56)
- Number slides from… (0 to 9999)
- Slide orientation
- Notes orientation



For our purposes, "independent" means that the value of one variable doesn't limit the values you can enter into the other variable.

Looking ahead at multi-dimensional variables

Non-independence example:

- Is 31 a valid day of the month?
  - December 31 is valid
  - February 31 is not

Components constrain each other:

- Date field (Year – Month – Day)
  - Some years have 366 days, most have 365
  - Some months have
    - 31 days
    - 30 days
    - 29 days
    - 28 days

You need a sampling strategy—we can't test all possible dates (and the consequences of selecting each date)—but simplistic testing at the boundaries won't work.

SUMMARY: A SCHEMA FOR DOMAIN TESTING

1. CHARACTERIZE THE VARIABLE
   - Identify the potentially interesting variables.
   - Identify the variable(s) you can analyze now. This is the variable(s) of interest.
   - Determine the primary dimension of the variable of interest.
   - Determine the type and scale of the variable's primary dimension and what values it can take.
   - Determine whether you can order the variable's values (from smallest to largest).
   - Determine whether this is an input variable or a result (or both).
   - Determine how the program uses this variable.
   - Determine whether other variables are related to this one.

From Kaner, Hoffman & Padmanabhan (2012). Domain Testing: A Workbook.

SUMMARY: A SCHEMA FOR DOMAIN TESTING

2. ANALYZE THE VARIABLE AND CREATE TESTS
   - Partition the variable (its primary dimension)
     - If the dimension is ordered, determine its sub-ranges and transition points.
     - If the dimension is not ordered, base partitioning on similarity.
   - Lay out the analysis in the classical boundary / equivalence class table. Test with best representatives.
   - Create tests for the consequences of the data entered, not just the input filter.
   - Identify secondary dimensions. Analyze them in the classical way.
   - Summarize your analysis with a risk / equivalence table.

From Kaner, Hoffman & Padmanabhan (2012). Domain Testing: A Workbook.

SUMMARY: A SCHEMA FOR DOMAIN TESTING

3. GENERALIZE TO MULTIDIMENSIONAL VARIABLES

- Analyze independent variables that should be tested together.
- Analyze variables that hold results.
- Analyze non-independent variables. Deal with relationships and constraints.

From Kaner, Hoffman & Padmanabhan (2012). Domain Testing: A Workbook.

SUMMARY: A SCHEMA FOR DOMAIN TESTING

4. PREPARE FOR ADDITIONAL TESTING
    - Identify and list unanalyzed variables. Gather information for later analysis.
    - Imagine (and document) risks that don't necessarily map to an obvious dimension.

From Kaner, Hoffman & Padmanabhan (2012). Domain Testing: A Workbook.

CLOSING THOUGHTS

- Domain analysis is a sampling strategy to cope with the problem of too many possible tests.
- Traditional domain analysis considers numeric input and output fields.
- Boundary analysis is optimized to expose a few types of errors such as miscoding of boundaries or ambiguity in definition of the valid/invalid sets.
  - However, there are other possible errors that boundary tests are insensitive to.
- The underlying concepts are simple.
- When you apply the concepts, domain testing starts out straightforward, but anything beyond the basics requires judgment.
  - When you say that this is equivalent to that, that's a judgment call on your part. They are probably equivalent in some ways and not equivalent in others.
- To a large degree, your decisions about equivalence, and your selection of specific values to test will be driven by the risk you want to explore about how the program might fail and how these variables might help you make the program fail in this way.

Black Box Software Testing: Introduction to Test Design: Lecture 6: Multivariable Testing

Cem Kaner, J.D., Ph.D.
Professor of Software Engineering: Florida Tech

Rebecca L. Fiedler, M.B.A., Ph.D.
President: Kaner, Fiedler & Associates

These notes are partially based on research that was supported by NSF Grants EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers" and CCLI-0717613 "Adaptation & Implementation of an Activity-Based Online or Hybrid Course in Software Testing." 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.

OVERVIEW OF THE COURSE

	Technique	Context / Evaluation
1	Function testing & tours.	A taxonomy of test techniques.
2	Risk-based testing, failure mode analysis and quicktests	Testing strategy. Introducing the Heuristic Test Strategy Model.
3	Specification-based testing.	(… work on your assignment …)
4	Use cases and scenarios.	Comparatively evaluating techniques.
5	Domain testing: traditional and risk-based	When you enter data, any part of the program that uses that data is a risk. Are you designing for that?
6	Testing combinations of independent and interacting variables.	Combinatorial, scenario-based, risk-based and logical-implication analyses of multiple variables.

LECTURE 6 READINGS

Required reading

- Czerwonka, J. (2008), Pairwise testing in the real world: Practical extensions to test-case scenarios. http://msdn.microsoft.com/en-us/library/cc150619.aspx

Recommended reading

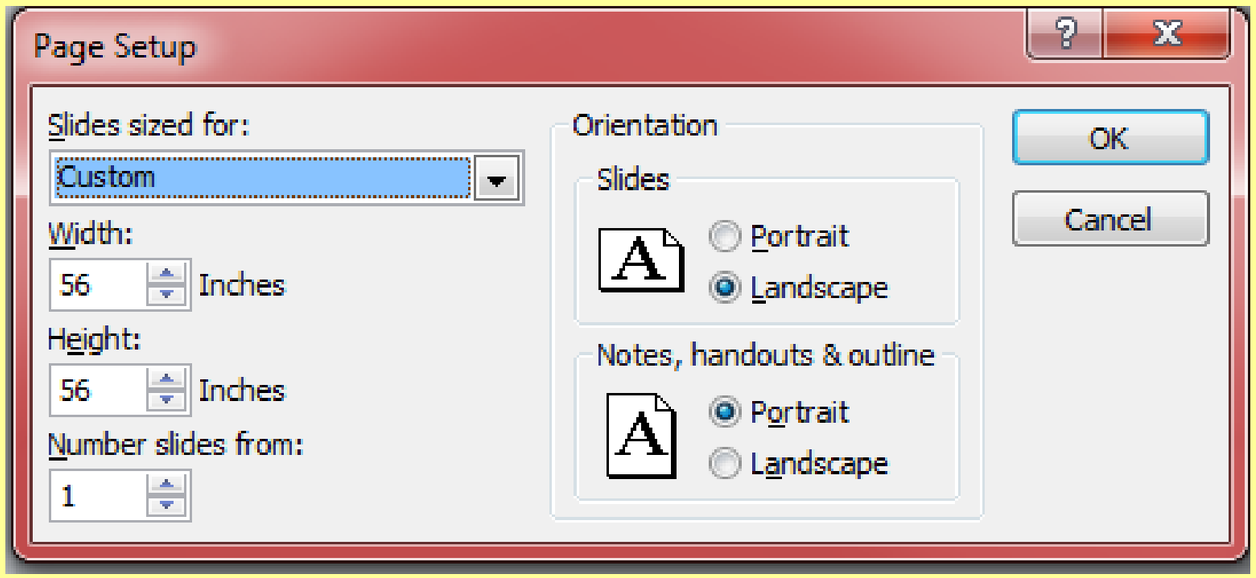
- Bach, J., and P. Schroeder (2004), "Pairwise Testing: A Best Practice that Isn't." Proceedings of the 22nd Pacific Northwest Software Quality Conference, 180–196. http://www.testingeducation.org/wtst5/PairwisePNSQC2004.pdf
- Bolton, M. (2007). Pairwise testing. http://www.developsense.com/pairwiseTesting.html
- Cohen, D. M., Dalal, S. R., Fredman, M. L., & Patton, G. C. (1997). The AETG system: An approach to testing based on combinatorial design. IEEE Transactions on Software Engineering, 23(7). http://aetgweb.argreenhouse.com/papers/1997-tse.html
- Microsoft's PICT tool is at http://download.microsoft.com/download/f/5/5/f55484df-8494-48fa-8dbd-8c6f76cc014b/pict33.msi.
- For more references, see http://www.pairwise.org/

Your class might use other readings, but we had these in mind when creating this lecture.

Independent variables: The Page Setup example

The PowerPoint Page Setup dialog shows several independent variables, including:

- Page width (1 to 56)
- Page Height (1 to 56)
- Number slides from… (0 to 9999)



For our purposes, "independent" means that the value of one variable doesn't limit the values you can enter into the other variable.

What should you test together, and why?

- Before testing variables in combination, test them individually
- Why bother to test them together?
  - Unexpected constraints on what you can enter
  - Unexpected consequences of the combination

WHAT VALUES SHOULD YOU TEST?

Remember from BBST Foundations:

Consider variables V_1, V_2, …, V_k

where

V_1 has N_1 possible values,

V_2 has N_2 possible values, etc.

and the variables are all independent.

The number of tests of combinations of the V_i's is

$N_1 * N_2 * \ldots * N_k$

What values should you test?

In this case, there are

- 5601 possible values of Page Width
- 5601 possible values of Page Height
- 10000 possible values of starting page number

  = 5601 * 5601 * 10000

  = 313,712,010,000 possible tests

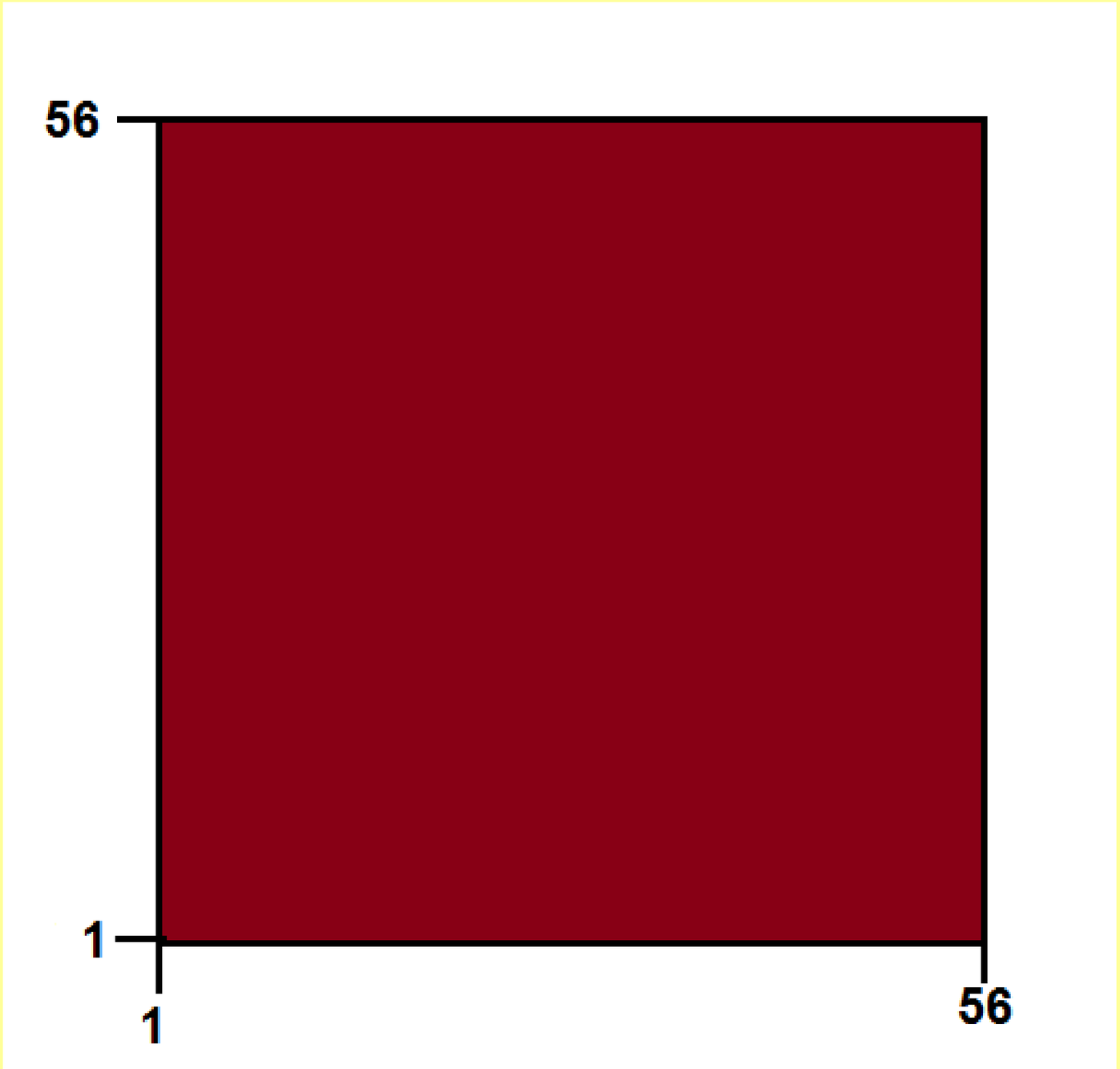
With so many possibilities, you have to select your tests with care.

What values should you test?

- The last calculation ignored invalid values.
- Typically:
  - Test invalid values in tests of the individual variables and don't test them in combination testing, or
  - Include a **few** tests with invalid values, each with a carefully chosen set of values to maximize the chance of exposing a suspected error.

In this section of this course, we will ignore invalid values.

Page width & Page height



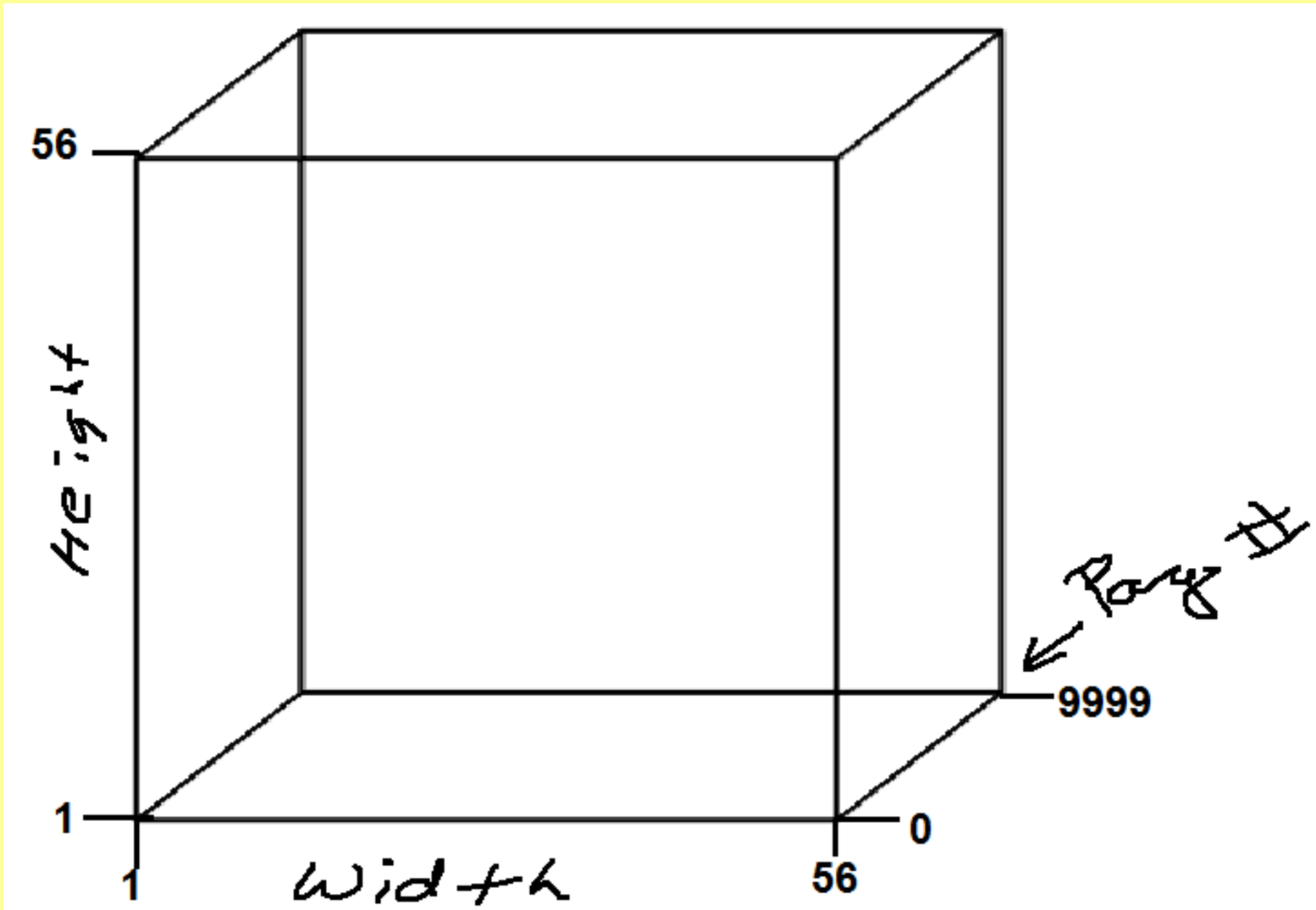
- 1 ≤ Height ≤ 56
- 1 ≤ Width ≤ 56

All values on the square or inside it are valid page sizes.

With *independent variables*, test at intersections of boundaries (the corners):

(1, 1), (1, 56), (56, 1), (56, 56)

PAGE WIDTH, PAGE HEIGHT, & PAGE NUMBER.



With three independent variables, the cube shows the valid domain.

The 8 corners are the usual valid-boundary tests.

Combination chart

	Width	Height	Page #
Test 1	1	1	1
Test 2	1	1	9999
Test 3	1	56	1
Test 4	1	56	9999
Test 5	56	1	1
Test 6	56	1	9999
Test 7	56	56	1
Test 8	56	56	9999

In a combination test, test several variables together. Each test explicitly sets values for each variable under test.

The combination chart shows the variable settings for each test.

Combination coverage: All singles

	Width	Height	Page #
Test 1	1	1	1
Test 2	56	56	9999

The All Singles coverage criterion is met if you include each value of each variable in at least one test.

README (note added in final edit…):
The Page # fields in these slides should run from 0 to 9999, not 1 to 9999. Apologies for any confusion…

Combination coverage: All Pairs

	Width	Height	Page #
Test 1	1	1	1
Test 2	1	56	9999
Test 3	56	1	9999
Test 4	56	56	1

The All Pairs coverage criterion is met if you include each pair of values of each pair of variables in at least one test.

Combination coverage: All Triples

	Width	Height	Page #
Test 1	1	1	1
Test 2	1	1	9999
Test 3	1	56	1
Test 4	1	56	9999
Test 5	56	1	1
Test 6	56	1	9999
Test 7	56	56	1
Test 8	56	56	9999

The All Triples coverage criterion is met if you include each 3-tuple of values of each group of 3 variables in at least one test.

Combination coverage: All N-tuples

	Width	Height	Page #
Test 1	1	1	1
Test 2	1	1	9999
Test 3	1	56	1
Test 4	1	56	9999
Test 5	56	1	1
Test 6	56	1	9999
Test 7	56	56	1
Test 8	56	56	9999

If you test N variables together, you meet the All N-tuples criterion by including every possible combination of the variables.

Configuration testing: Independent variables

Classic example of combination testing:

- O/S: Windows 7, VISTA, XP
- Printer: HP, Epson, Lexmark
- Memory: Low, Medium, High
- Processor: 1-core, 2-core, 4-core
- Graphics: Slow, medium, fast
- Hard drive: 0 drives, 1 drive, 2 drives

- Number of possible tests =

  3 * 3 * 3 * 3 * 3 * 3 = 729

Setting up for combination testing (if you're creating the combination table by hand)

1. Select the variables to test
2. Select the test values for each variable
   - You want the smallest reasonable set for each variable because you are multiplying the numbers
3. Assign 1-character abbreviations for each value of each variable, to make the chart simple
4. Decide on your coverage criterion
5. Create the combination chart

In the config test example, we've done the first two steps already.

Abbreviations of our variables

O/S:

1 Windows 7

2 VISTA

3 XP

Processor:

1 1-core

2 2-core

3 4- core

Printer:

1 HP

2 Epson

3 Lexmark

Graphics:

1 Slow

2 Medium

3 Fast

Memory:

1 Low

2 Medium

3 High

Hard drive:

1 0 drives

2 1 drive

3 2 drives

Setting up for combination testing (if you're creating the combination table by hand)

1. Select the variables to test
2. Select the test values for each variable
   - You want the smallest reasonable set for each variable because you are multiplying the numbers
3. Assign 1-character abbreviations for each value of each variable, to make the chart simple
4. Decide on your coverage criterion
5. Create the combination chart

In the config test example, we've done the first two steps already.

All Singles

Test

- every value
  - (every value you decided to test)
- of each variable
- at least once.

	O/S	Printer	Memory	Processor	Graphics	Drive
Test 1	1	1	1	1	1	1
Test 2	2	2	2	2	2	2
Test 3	3	3	3	3	3	3

All Pairs

Add one variable to the table at a time.

Sort the variables

- put the variable with the most values in the first column
- the second-most values in the second column
- For example, if you were testing 4 types of printers (and stayed with 3 of everything else), we'd add printers first.

All Pairs

In this case, each variable has 3 values.

The number of pairs of the first two values is 3 * 3 = 9

	O/S	Printer	Memory	Processor	Graphics	Drive
Test 1	1	1				
Test 2	1	2				
Test 3	1	3				
Test 4	2	1				
Test 5	2	2				
Test 6	2	3				
Test 7	3	1				
Test 8	3	2				
Test 9	3	3				

All Pairs

Add the third variable

	O/S	Printer	Memory	Processor	Graphics	Drive
Test 1	1	1	1			
Test 2	1	2	2			
Test 3	1	3	3			
Test 4	2	1	2			
Test 5	2	2	3			
Test 6	2	3	1			
Test 7	3	1	3			
Test 8	3	2	1			
Test 9	3	3	2			

All Pairs

Add the fourth variable

	O/S	Printer	Memory	Processor	Graphics	Drive
Test 1	1	1	1	1		
Test 2	1	2	2	2		
Test 3	1	3	3	3		
Test 4	2	1	2	3		
Test 5	2	2	3	1		
Test 6	2	3	1	2		
Test 7	3	1	3	2		
Test 8	3	2	1	3		
Test 9	3	3	2	1		

All Pairs

	O/S	Printer	Memory	Processor	Graphics	Drive
Test 1	1	1	1	1	1	
Test 2	1	2	2	2	2	
Test 3	1	3	3	3	3	
Test 4	2	1	2	3	1	
Test 5	2	2	3	1	3	
Test 6	2	3	1	2	2	
Test 7	3	1	3	2	2	
Test 8	3	2	1	3	1	
Test 9	3	3	2	1	3	
Test 10						
Test 11						
Test 12						
Test 13						
Test 14						

We'll need more rows for a fifth variable

Start by checking for all-pairs in the first and fifth columns.

No extra tests are needed (yet)

All Pairs

	O/S	Printer	Memory	Processor	Graphics	Drive
Test 1	1	1	1	1	1	
Test 2	1	2	2	2	2	
Test 3	1	3	3	3	3	
Test 4	2	1	2	3	1	
Test 5	2	2	3	1	3	
Test 6	2	3	1	2	2	
Test 7	3	1	3	2	2	
Test 8	3	2	1	3	1	
Test 9	3	3	2	1	3	
Test 10		1			3	
Test 11		3			1	
Test 12						
Test 13						
Test 14						

Now take care of the second and fifth columns.

I'm showing the values you need for testing. You can fill any other values in Tests 10 and 11 and achieve coverage.

All Pairs

	O/S	Printer	Memory	Processor	Graphics	Drive
Test 1	1	1	1	1	1	
Test 2	1	2	2	2	2	
Test 3	1	3	3	3	3	
Test 4	2	1	2	3	1	
Test 5	2	2	3	1	3	
Test 6	2	3	1	2	2	
Test 7	3	1	3	2	2	
Test 8	3	2	1	3	1	
Test 9	3	3	2	1	3	
Test 10		1	1		3	
Test 11		3	3		1	
Test 12						
Test 13						
Test 14						

Third column and fifth

All Pairs

	O/S	Printer	Memory	Processor	Graphics	Drive
Test 1	1	1	1	1	1	
Test 2	1	2	2	2	2	
Test 3	1	3	3	3	3	
Test 4	2	1	2	3	1	
Test 5	2	2	3	1	3	
Test 6	2	3	1	2	2	
Test 7	3	1	3	2	2	
Test 8	3	2	1	3	1	
Test 9	3	3	2	1	3	
Test 10		1	1	2	3	
Test 11		3	3	2	1	
Test 12				1	2	
Test 13				3	2	
Test 14						

Fourth column and fifth

All Pairs

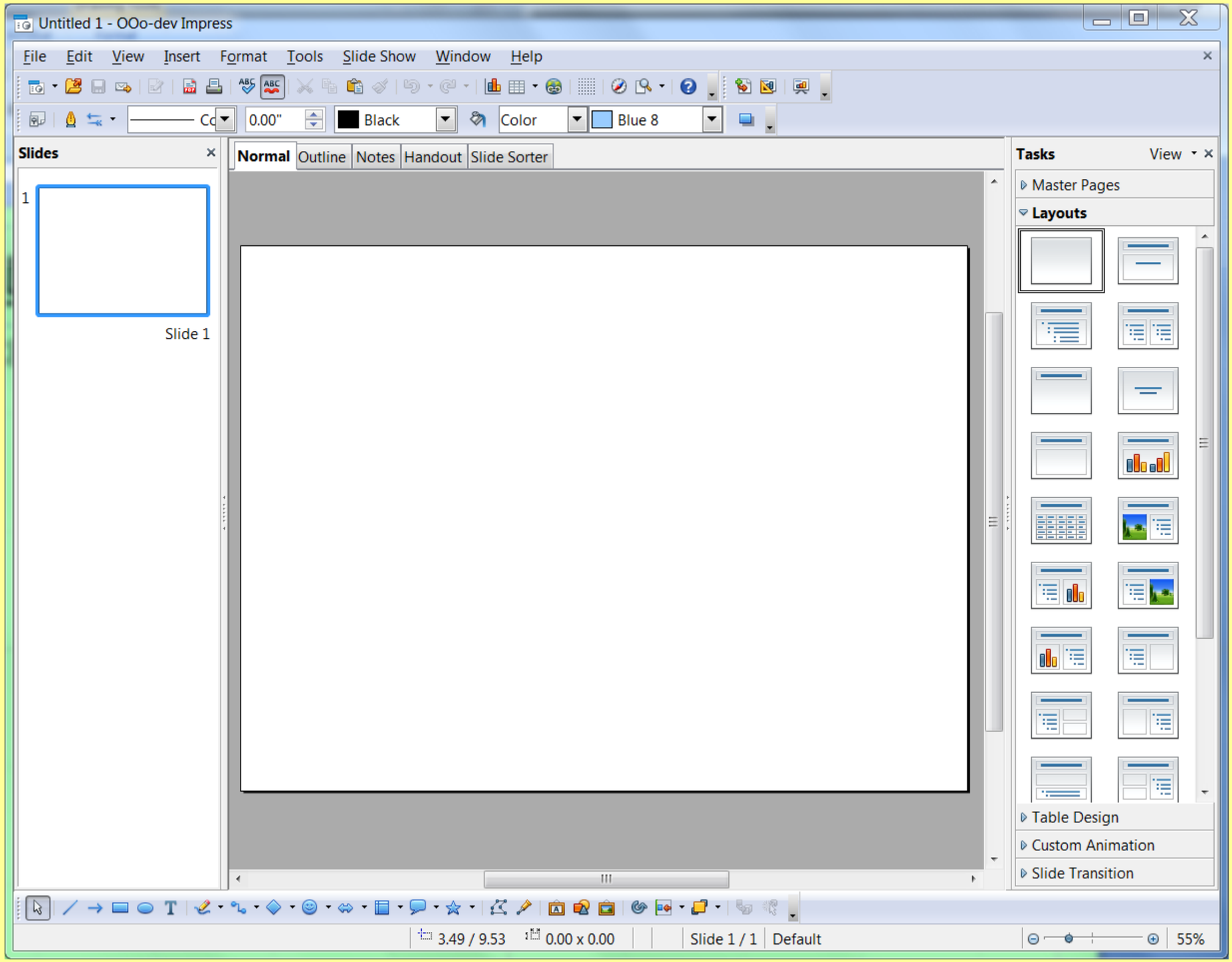
	O/S	Printer	Memory	Processor	Graphics	Drive
Test 1	1	1	1	1	1	1
Test 2	1	2	2	2	2	2
Test 3	1	3	3	3	3	3
Test 4	2	1	2	3	1	2
Test 5	2	2	3	1	3	1
Test 6	2	3	1	2	2	3
Test 7	3	1	3	2	2	3
Test 8	3	2	1	3	1	3
Test 9	3	3	2	1	3	2
Test 10		1	1	2	3	2
Test 11	3	3	3	2	1	1
Test 12		2	2	1	2	3
Test 13			2	3	2	1
Test 14			3			2

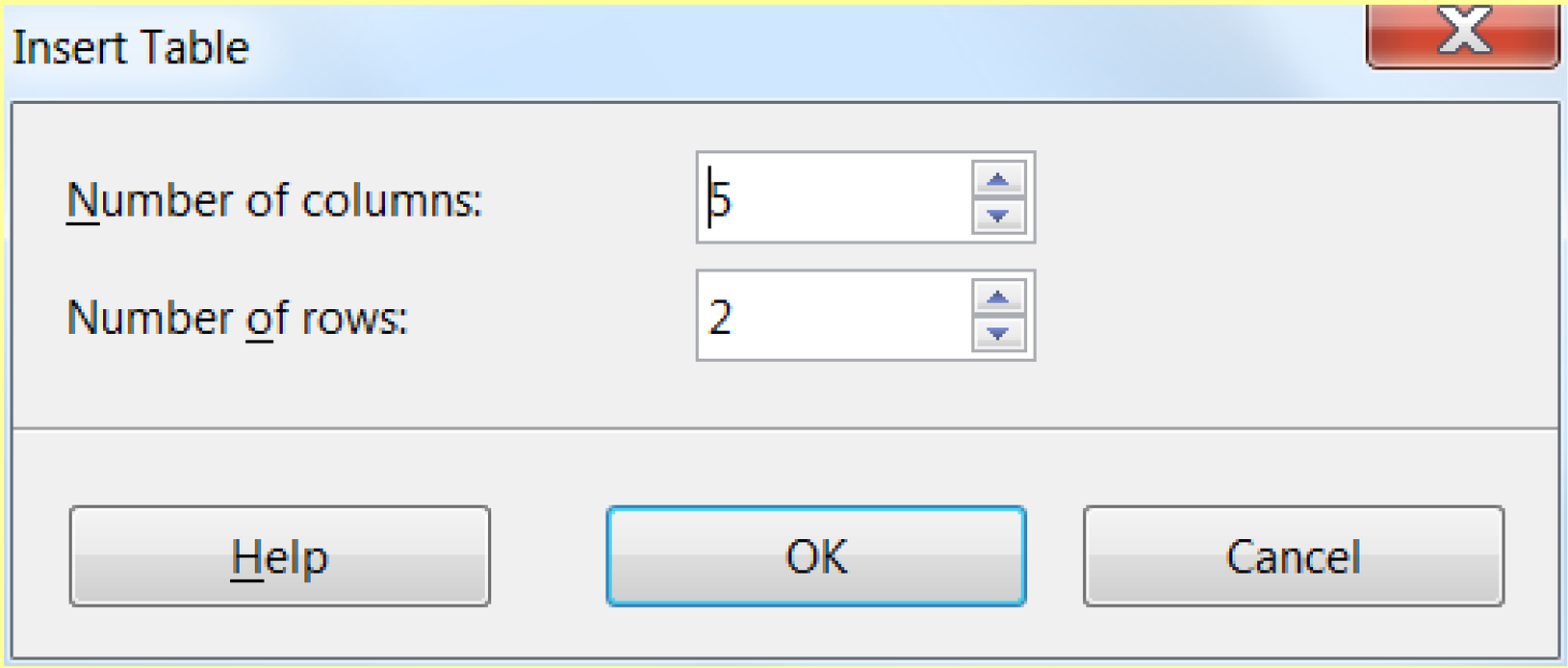
And here's the 6th variable.

The more you have, the harder it gets, but the greater the savings compared to the total number of combinations.

Another all-pairs example

Greetings from Open Office Impress



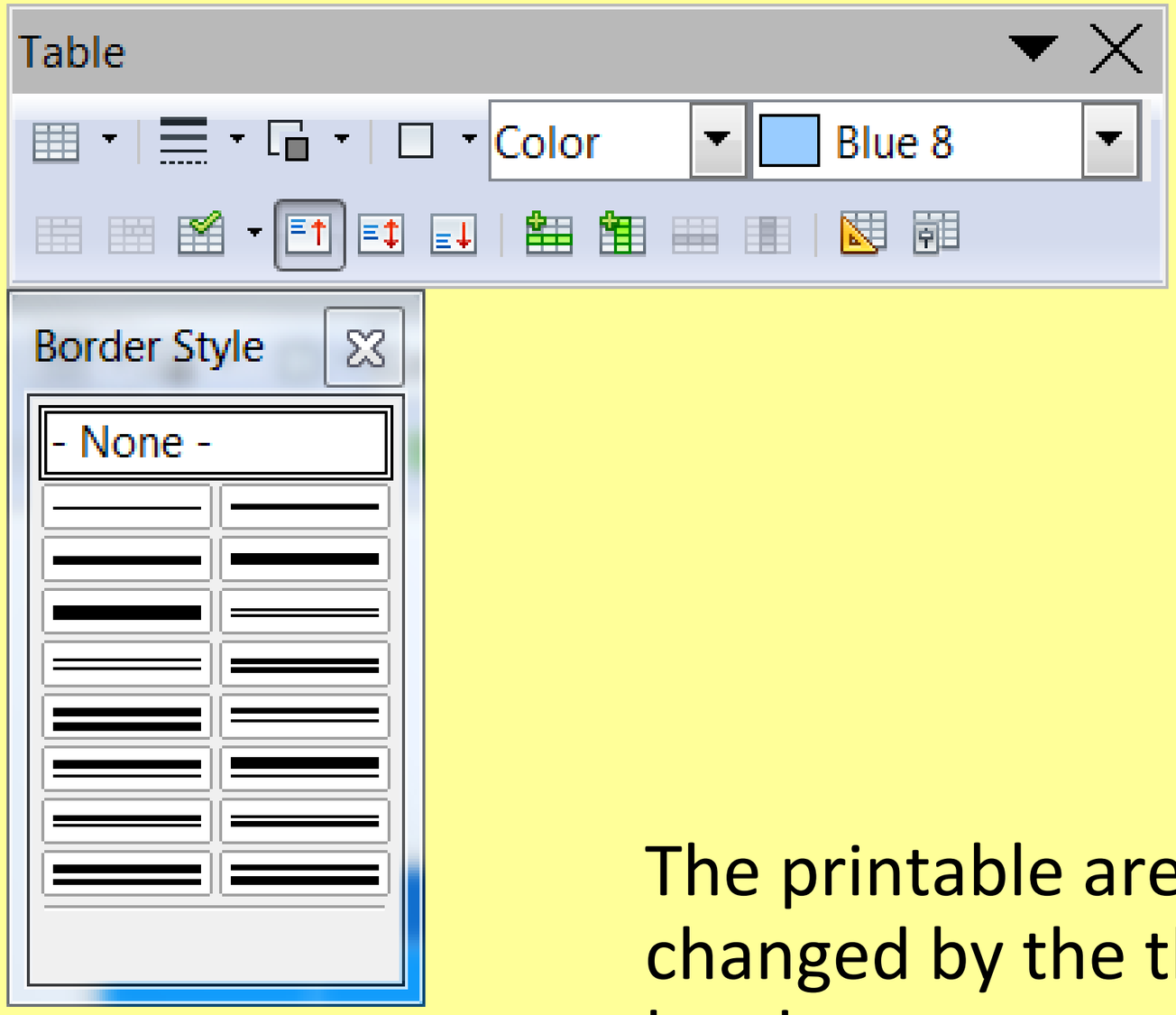


Let’s put a table on the slide. What can you do with a table?

- You can add up to 75 rows and up to 75 columns.
- What can you fit in these rows and columns?
- You can specify the height and width of the cells.
  - The possible values depend on the number of rows/columns compared to the size of the page.

Setting up for combination testing (if you're creating the combination table by hand)

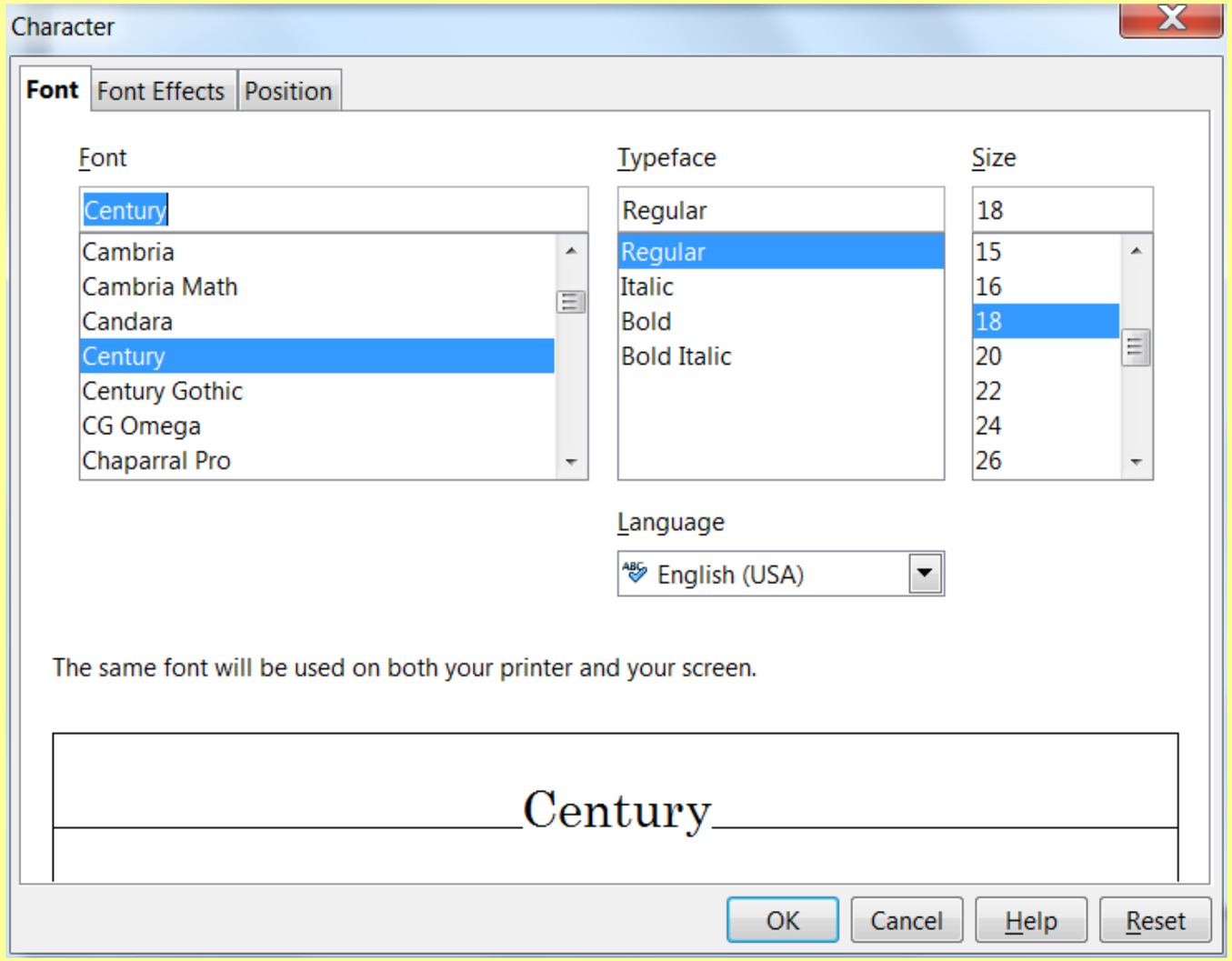
1. Select the variables to test
2. Select the test values for each variable
   - You want the smallest reasonable set for each variable because you are multiplying the numbers
3. Assign 1-character abbreviations for each value of each variable, to make the chart simple
4. Decide on your coverage criterion
5. Create the combination chart



The printable area within a cell is also changed by the thickness of the cell's border.

ADD SOME TEXT

- Sans Serif faces are plain, like this one.
- Serif typefaces have little doo-dads (called serifs). Some programs mis-estimate the size of a character with a serif, especially letters like W and *W.*
- If the W is too wide to fully fit in the cell, many programs will print placeholder text (like "…") instead of half the character.



COMBINATIONS

Now organize this to facilitate all-pairs combination testing…

• ROWS: 1 row, 75 rows	1, 75
• ROW HEIGHT: small, large, max possible for this table	S, L, M
• COLUMNS: 1 column, 75 columns	1, 75
• COLUMN WIDTH: small, large, max possible for this table	S, L, M
• BORDER: none, wide, max for this cell	N, W, M
• TYPEFACE: Arial (sans serif), Century (serif), Century italic	A, C, I
• TEXT SIZE: Barely fits, Barely too big, Way too big	F, T, W

CREATE THE COMBINATION TABLE

- Start by organizing the variables:
  - most values-to-test to fewest
- Set up the first pairs

	Row height	Column width	Border	Typeface	Text size	Rows	Cols
1	S	S					
2	S	L					
3	S	M					
4	L	S					
5	L	L					
6	L	M					
7	M	S					
8	M	L					
9	M	M					

Add the next variable

- Check to make sure that every value of this variable pairs with every value of every other variable

	Row height	Column width	Border	Typeface	Text size	Rows	Cols
1	S	S	N				
2	S	L	W				
3	S	M	M				
4	L	S	W				
5	L	L	M				
6	L	M	N				
7	M	S	M				
8	M	L	N				
9	M	M	W				

ADD THE NEXT VARIABLE

- Check to make sure that every value of this variable pairs with every value of every other variable

	Row height	Column width	Border	Typeface	Text size	Rows	Cols
1	S	S	N	A			
2	S	L	W	C			
3	S	M	M	I			
4	L	S	W	I			
5	L	L	M	A			
6	L	M	N	C			
7	M	S	M	C			
8	M	L	N	I			
9	M	M	W	A			

ADD THE NEXT VARIABLE

- We've exhausted the simple permutations and must add a few rows

	Row height	Column width	Border	Typeface	Text size	Rows	Cols
1	S	S	N	A	F		
2	S	L	W	C	T		
3	S	M	M	I	W		
4	L	S	W	I	F		
5	L	L	M	A	T		
6	L	M	N	C	W		
7	M	S	M	C	T		
8	M	L	N	I	W		
9	M	M	W	A	F		
10	S	L	M	C	F		
11	L	M	N	I	T		
12	M	S	W	A	W		

ADD THE NEXT VARIABLE

- The last two are easy…

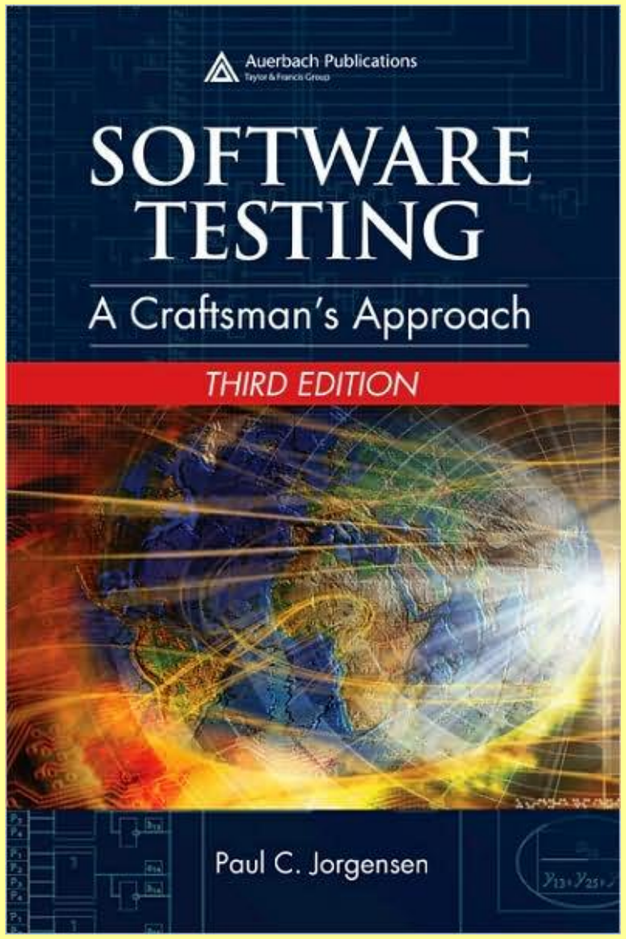
	Row height	Column width	Border	Typeface	Text size	Rows	Cols
1	S	S	N	A	F	1	1
2	S	L	W	C	T	1	75
3	S	M	M	I	W	75	1
4	L	S	W	I	F	75	75
5	L	L	M	A	T	75	1
6	L	M	N	C	W	1	75
7	M	S	M	C	T	75	1
8	M	L	N	I	W	1	1
9	M	M	W	A	F	75	75
10	S	L	M	C	F	1	75
11	L	M	N	I	T	75	1
12	M	S	W	A	W	1	1

A LITTLE TERMINOLOGY

SOME TERMINOLOGY

Jorgensen (2008) makes three distinctions that students often find helpful in puzzling through the coverage decisions made in a combination test:

- "Exhaustive" vs. "Equivalence class"
- "Robust" versus "Normal"
- "Strong" versus "Weak"



Terminology: Exhaustive versus Equivalence

Consider a single variable with one dimension and a simple class of valid values:

$$0 < X < 24.$$

Run two tests for every boundary:

- the boundary-valid values (24-Δ and 0+Δ)
- the boundary-invalid values (24 and 0)

You end up with 4 values:

- Too-low (TL) and valid lowest (VL)
- Too-big (TB) and valid biggest (VB) where

$$TL = VL- \Delta \text{ and } TB=VB+\Delta$$

Terminology: Exhaustive versus Equivalence

- In multi-dimensional testing, start by testing each dimension on its own, reasonably thoroughly.
- Then decide to either test
  - exhaustively (every value of the variable), or using only
  - equivalence class representatives (such as TL, VL, VB and TB)

In combination testing, when we say "combine every value of a variable", we mean every value that we have decided to test. This may be an exhaustive sample, the equivalence class subset, or some other defined list.

SOME TERMINOLOGY

- **"Normal"**: Includes only "valid" cases
- **"Robust"**: Includes error cases as well

I typically do "robust" testing of individual variables and "normal" testing of combinations.

SOME TERMINOLOGY

- **"Weak"**: Each value of each variable will appear in at least one combination. (All singles)
- **"Strong"**: Each value of each variable appears in combination with each other variable. (All N-tuples)

3-VARIABLE EXAMPLE

- Three numeric variables, V1and V2.
- Equivalence class and boundary analysis yields these test values
  - V1: TL, VL, VB and TB
  - V2: TL, VL, VB and TB
  - V3: TL, VL, VB and TB

TL: too low
VL: valid lowest
VB: valid biggest
TB: too big

Weak robust equivalence

	V1	V2	V3
Test 1	VL	VL	VL
Test 2	VB	VB	VB
Test 3	TL	TL	TL
Test 4	TB	TB	TB

- All singles, including invalid values
- Create enough tests to cover every value of every variable, once. If the largest number of values is N, you need only N tests
- Note the collisions of error cases. If Test 3 fails, is it because of the bad value of V1, V2, V3, or some combination of them?

What bug do you expect to find in Test 3 that you would not find in a test of single dimension, with a bad value? Why do you need a combination of invalid values?

Weak robust equivalence revised

	V1	V2	V3
Test 1	VL	VL	VL
Test 2	VB	VB	VB
Test 3	TL	VL	VB
Test 4	VB	TL	VL
Test 5	VL	VL	TL
Test 6	TB	VB	VL
Test 7	VL	TB	VB
Test 8	VB	VL	TB

Treat error cases specially:

- All-singles for “valid" (non-error) inputs
- Add tests that allow one error per test case.

Weak normal equivalence

	V1	V2	V3
Test 1	LV	LV	LV
Test 2	BV	BV	BV

All singles

- Only valid values
- Test invalid cases in single-variable tests, not combination tests.

Note the coverage that you do and do not achieve:

- You have a test for every valid value of interest of every variable
- You might catch some interactions among variables, but there is no coverage of interactions.

We often add a few market-critical combinations to an all-singles set of tests.

Strong Normal equivalence

	V1	V2	V3
Test 1	LV	LV	LV
Test 2	LV	LV	BV
Test 3	LV	BV	LV
Test 4	LV	BV	BV
Test 5	BV	LV	LV
Test 6	BV	LV	BV
Test 7	BV	BV	LV
Test 8	BV	BV	BV

- All N-tuples
- Valid values only
- Test error cases in one-variable tests
- If there are N independent dimensions, and you test only LV and BV for each, there are 2^N tests

A variable might have
- **more than one valid equivalence class**
- **and more than 2 values of interest.**

STRONG ROBUST EQUIVALENCE

- All N-tuples
- The table is too big to show here
- With N independent dimensions,
  - TL, LB, UB, TB yields 4 values per dimension
  - 4^N tests

Tests that include several errors are of interest only if you think that multiple errors might have some type of cumulative effect.

So, now

you know

what tests to run,

right?

(Well, maybe not…)

What are the risks?

What if memory management is a risk?
(It is…)

What tests do you need to run to understand the impact of table size (combined with table cell content) on memory management?

You have to study consequences in your testing, not just inputs.

CONSEQUENCES, CONSEQUENCES

- Discussions of combination testing typically focus on the variables you're going to test and the values you're going to test them with.
- But **what's the test?**
  - Setting the variables to their values is only the first step
  - Running some basic stability tests is unlikely to tell you much
- **What are the consequences of this combination?**

CONSEQUENCES, CONSEQUENCES

- For example, test a configuration that includes a new printer and a new video card. What should you test?
  - Basic printer functions
  - Basic display functions
  - But what involves printers **and** video?
    - e.g. print preview

CONSEQUENCES, CONSEQUENCES

- For example, test page layout with:
  - margins, headers and footers
- Big fonts might not be very interesting to test for most documents
  - but how will the program handle a one-character word in a font big enough to be bigger than the displayable area
    - because the margins are too wide, so space for text is narrow and
    - headers and footers are too tall, so space for text is short.

If you don't ask, as part of a combination test, what are the special risks posed by that particular combination, what are you really testing?

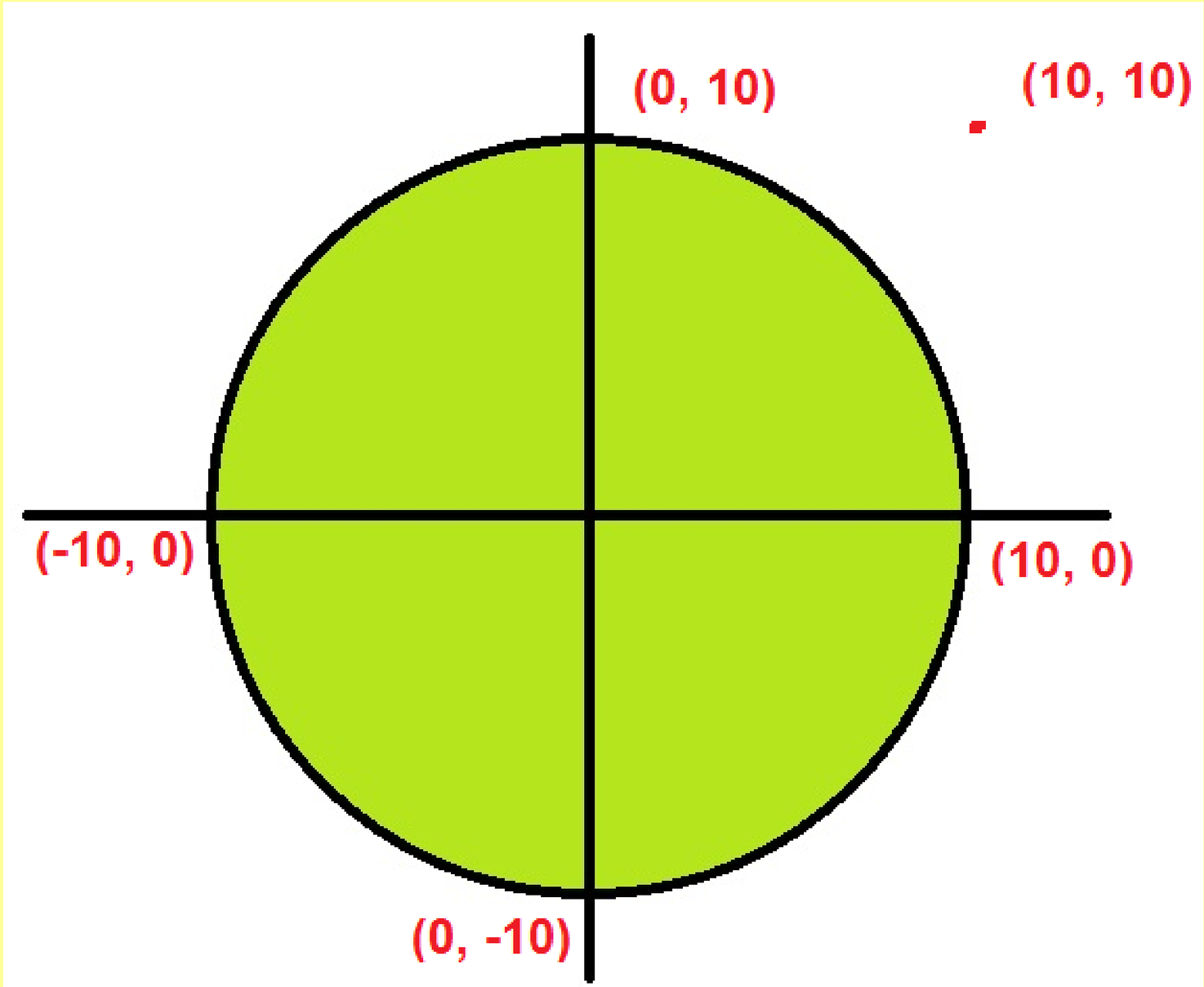
WORKING WITH VARIABLES THAT ARE NOT INDEPENDENT

Independent vs. non-independent

- All-pairs is useful for checking for relationships among variables that aren't supposed to be related (like printers and video cards).
- But how much testing time do you want to spend confirming that:
  - things that aren't supposed to be related aren't related?
- Don't you also want to test:
  - the things that **are** supposed to be related, to see
  - whether the relationship is implemented correctly, or
  - whether your model of the relationship is correct?

Jorgensen (2008) argues that combinatorial approaches like all-pairs have been over-promoted and provide less value than some people expect.

What if they aren't independent?



$-10 \leq X \leq 10$

$-10 \leq Y \leq 10$

$X^2 + Y^2 \leq 100$

When the value of one variable constrains another, corner cases like (10,10) probably aren't of much interest.

In this case, the border is the circle.

Another variable with components that constrain each other: Date.

28 days	February not a leap year
29 days	February leap year
30 days	January, March, May, July, August, October, December
31 days	April, June, September, November

Date field (Year – Month – Day)

- (2013 – 2 – 28) is a valid boundary
- (2013 – 2 – 29) is an invalid boundary

See Jorgensen (2008) for a detailed analysis

You need a sampling strategy—we can't test all possible dates (and the consequences of selecting each date)—but simplistic testing at the boundaries won't work.

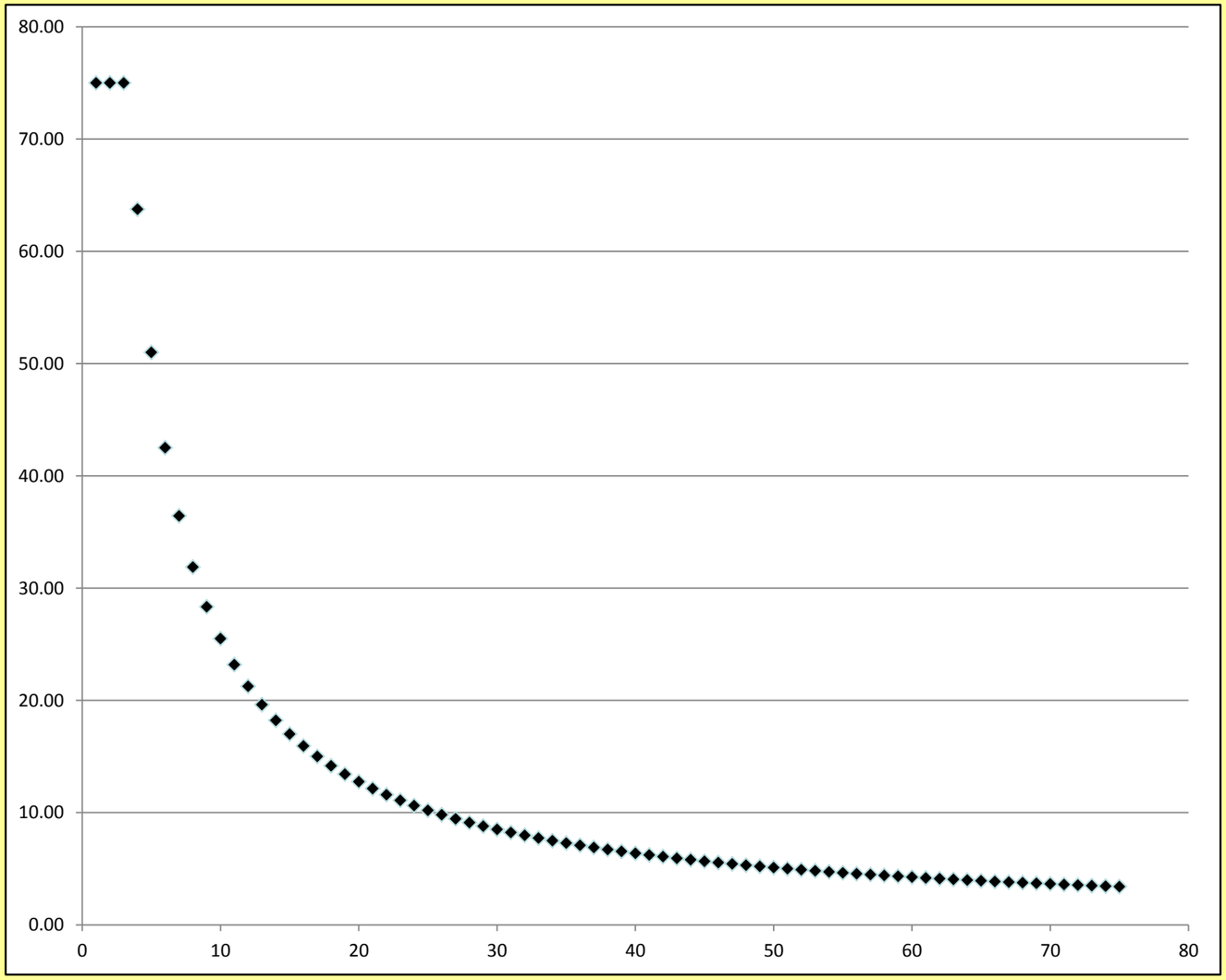
Back to the Open Office table

	Row height	Column width	Border	Typeface	Text size	Rows	Cols
1	S	S	N	A	F	1	1
2	S	L	W	C	T	1	75
3	S	M	M	I	W	75	1
4	L	S	W	I	F	75	75
5	L	L	M	A	T	75	1
6	L	M	N	C	W	1	75
7	M	S	M	C	T	75	1
8	M	L	N	I	W	1	1
9	M	M	W	A	F	75	75
10	S	L	M	C	F	1	75
11	L	M	N	I	T	75	1
12	M	S	W	A	W	1	1

THESE ARE NOT INDEPENDENT

- (75 rows, 75 columns) is impossible.
- You can have (1, 75) or (75, 1), but in OpenOffice, rows * columns must yield fewer than 255 cells.
- Here's a graph of the possible values:

$$R \leq 75$$
$$C \leq 75$$
$$R * C \leq 255$$



More non-independence

- Border size, text size and column width all constrain each other.
- You can often work around the assumption of independence and still achieve "all pairs" by adding somewhat-redundant tests.
- However, the more related the variables, the many more tests you have to create.

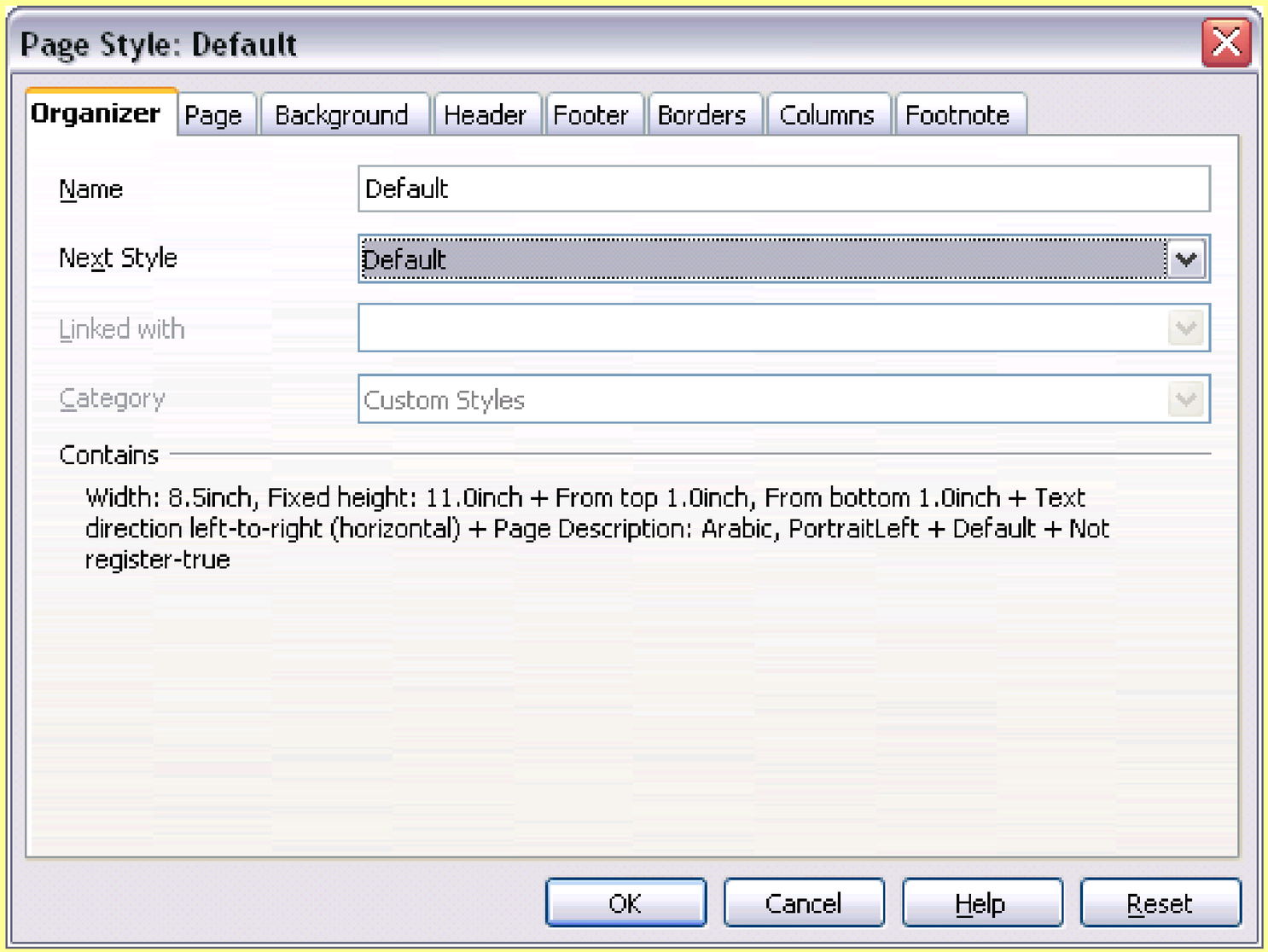
Open Questions

All-pairs combination testing doesn't tell you:

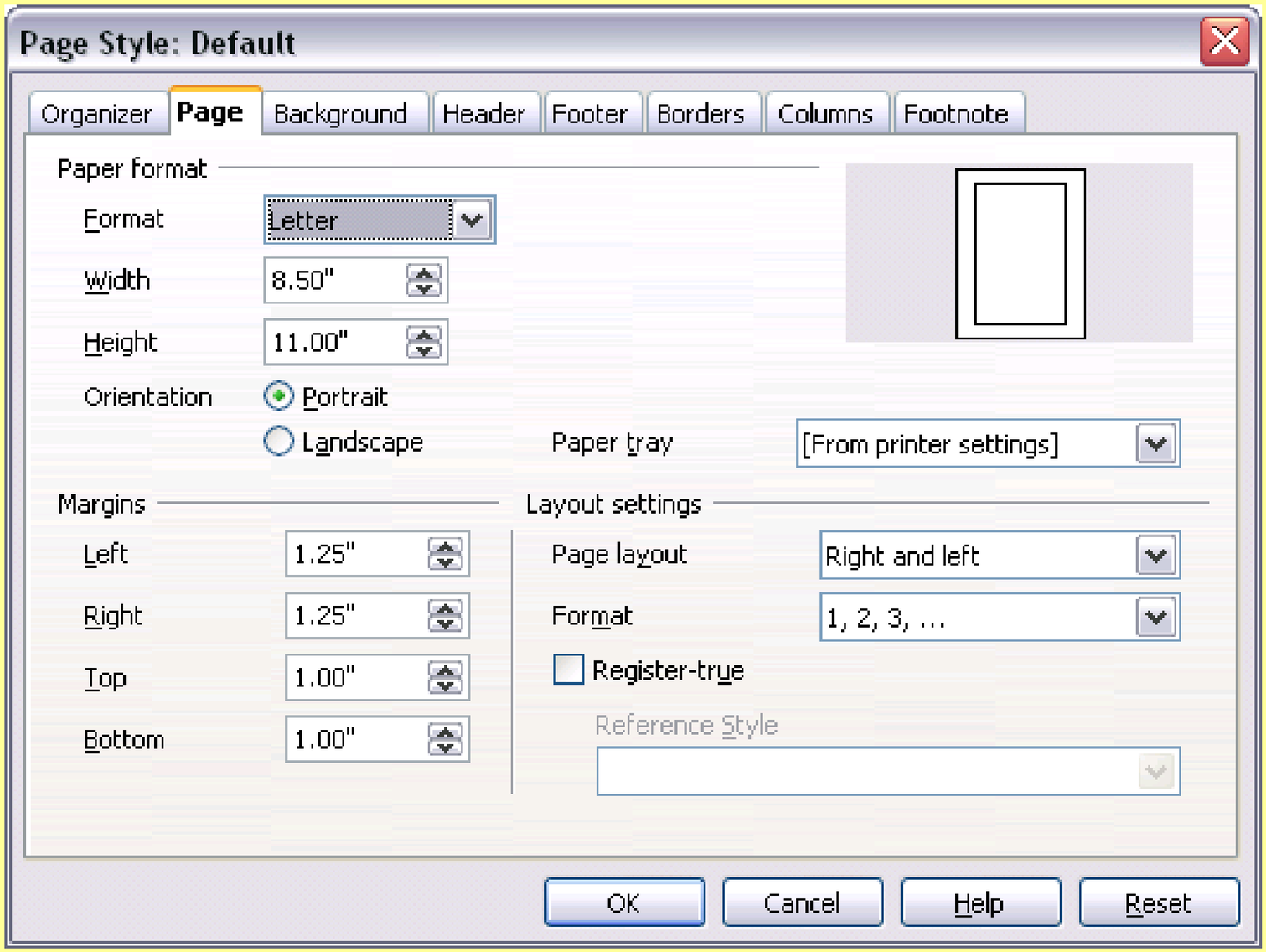
- Which variables you should test together, and why
- Which values of these variables you should test, and why
- How you should deal with relationships among the variables
- What risks you should look for when testing these variables together
- How you should determine whether the program passed or failed the test

Interdependence of several variables

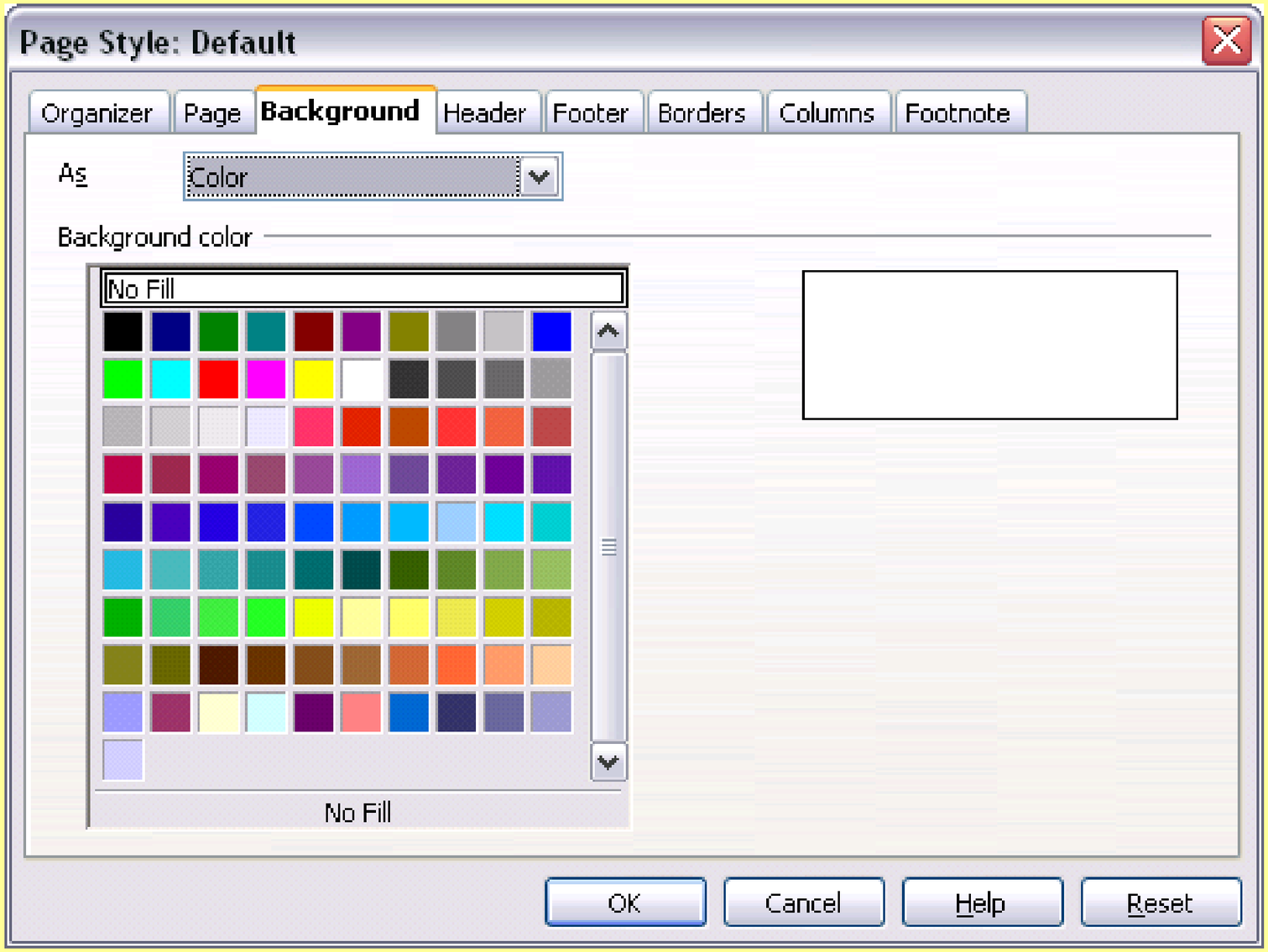
Here is the OpenOffice page style dialog. All of these variables interact in the layout of the page.



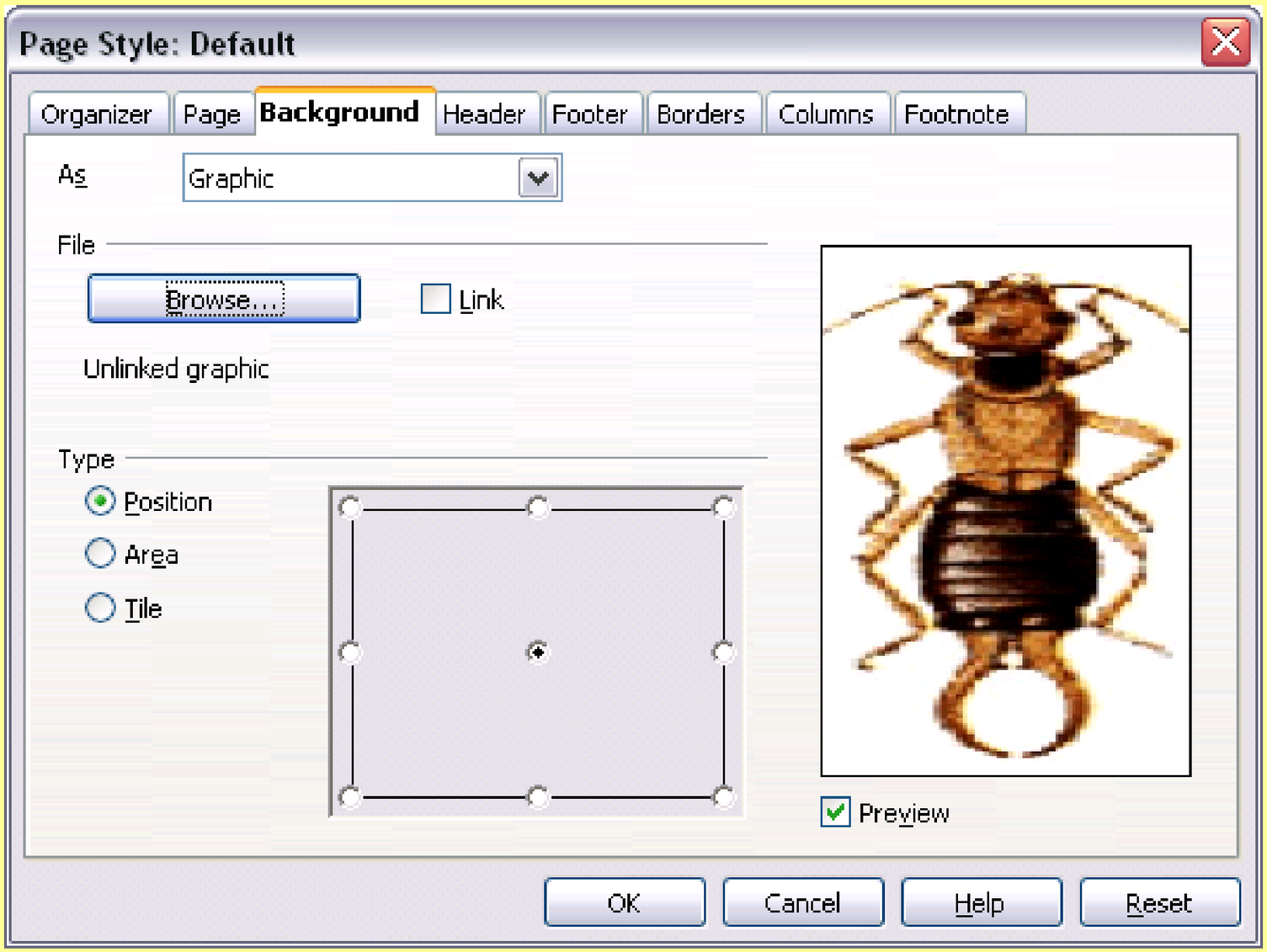
PAGE SIZE AND MARGINS



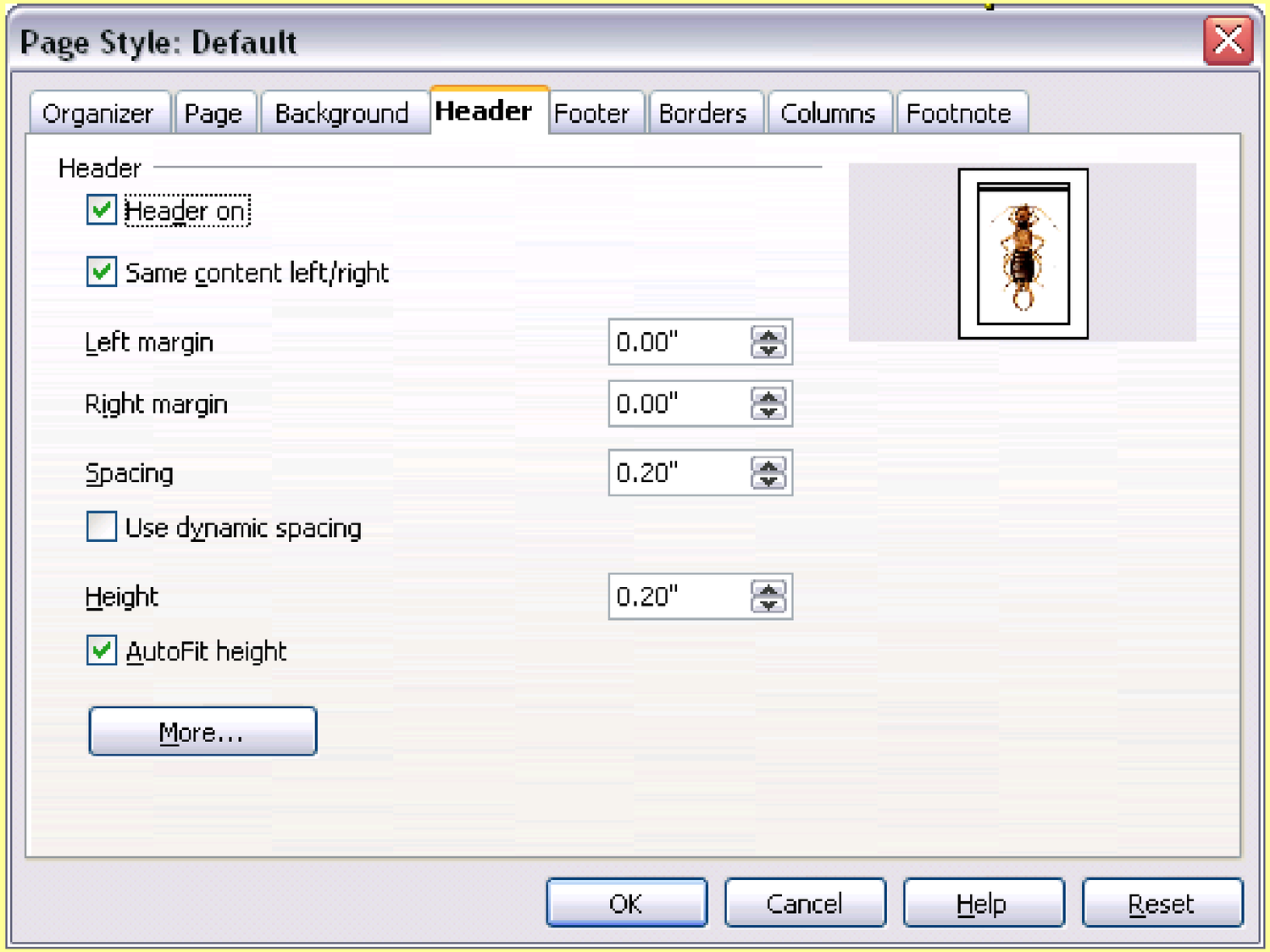
Background color or graphics



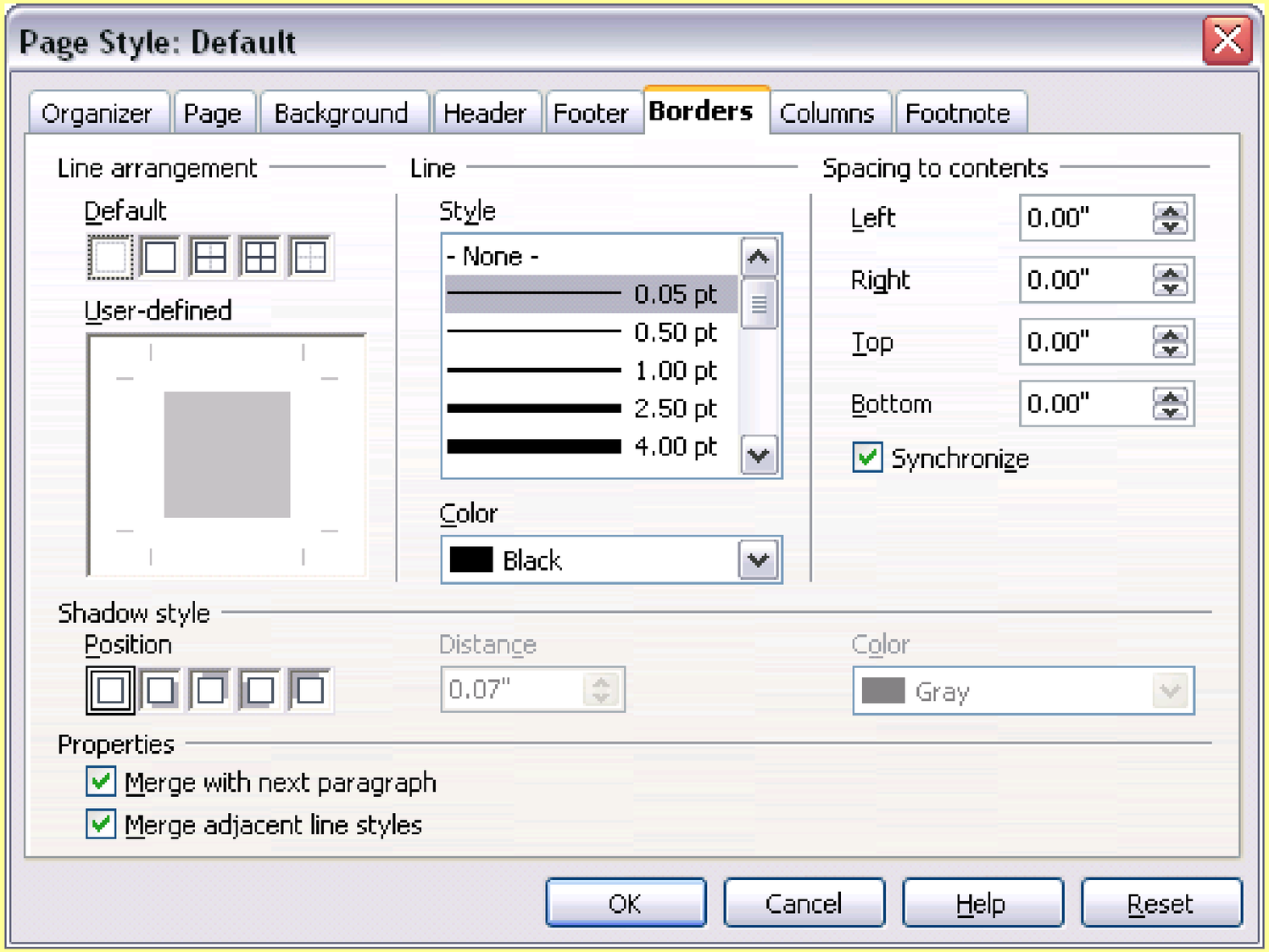
BACKGROUND GRAPHICS ARE CONSTRAINED BY PAGE DIMENSIONS.



Can you list the relevant variables?

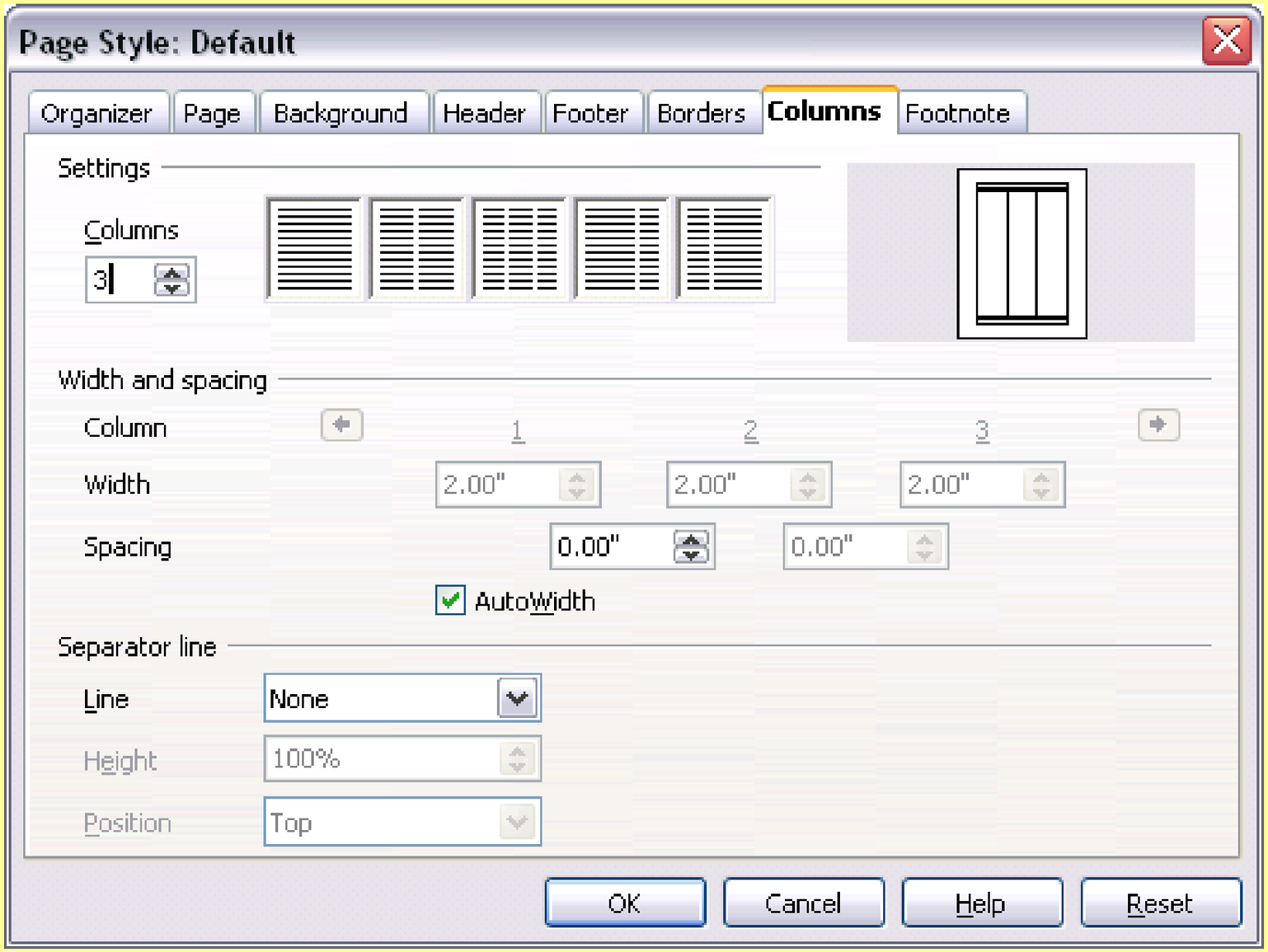


How many variables are on this page?

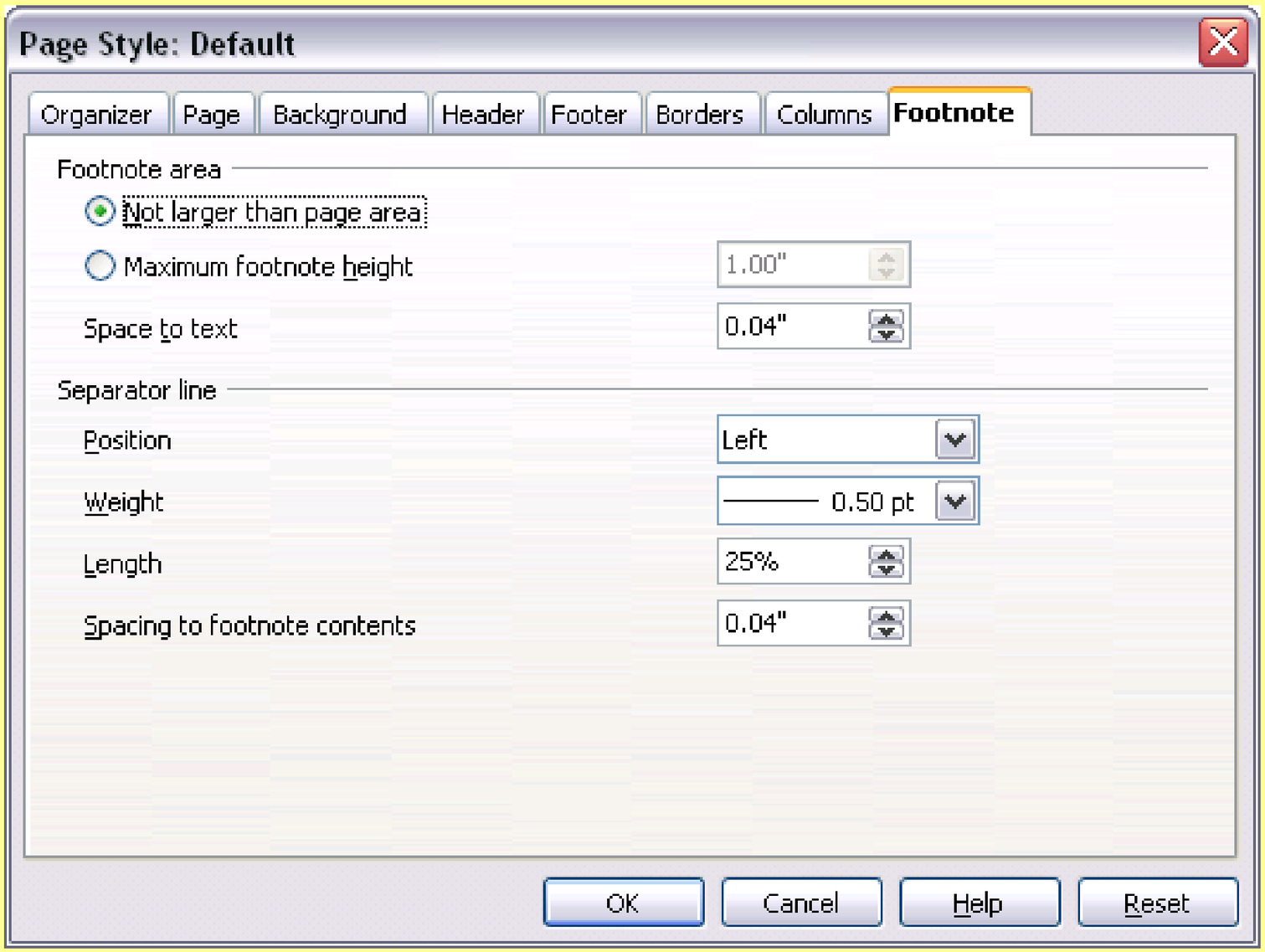


How do borders interact with background graphics? Hmmm…

THE NUMBER OF VARIABLES ON THIS PAGE DEPENDS ON HOW MANY COLUMNS YOU CHOOSE



Last panel



I don't know how to do all pairs on a dialog like this.

Exploring related variables

Try a variable relationship tour to identify and characterize relationships.

For each variable:

- 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

VARIABLE RELATIONSHIPS TABLE

FIELD	ENTRY SOURCE	DISPLAY	PRINT OR SAVE	RELATED VARIABLE	RELATIONSHIPS
Variable 1	Every way you can change values in V1	Every way you can display V1	Every way you can print, transfer or store V1	Variable 2	V1 < V2 (Constraint to a range)
Variable 2	Every way you can change values in V2			Variable 1	V2 > V1 (Constraint to a range)

MULTIVARIABLE RELATIONSHIPS

Two variables, **V1** and **V2**

V1 = f(V2)

or **V1 = f(V2, V3, V4, ...)**

or **V1 is constrained by V2**

Constraint examples:

- **V1 < V2+K**
- **V1** is an enumerated variable. 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**)

Multivariable relationships

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.

VARIABLE RELATIONSHIPS

If you can break a constraint, exploit it.

If you can get the program to accept

V1 = 100 and V2 = 20

Try displaying, printing or saving these values or using them in some other way.

Follow-on failures can be more persuasive than a "mere" constraint violation.

FIELD	ENTRY SOURCE	DISPLAY	PRINT OR SAVE	RELATED VARIABLE	RELATIONSHIPS
Variable 1	Every way you can change values in V1	Every way you can display V1	Every way you can print, transfer or store V1	Variable 2	V1 < V2 (Constraint to a range)
Variable 2	Every way you can change values in V2			Variable 1	V2 > V1 (Constraint to a range)

Approaches to combination testing

- ***Mechanical (or procedural).*** The tester uses a routine procedure to determine a good set of tests
- ***Risk-based.*** The tester combines test values (the values of each variable) based on perceived risks associated with noteworthy combinations
- ***Scenario-based.*** The tester combines test values on the basis of interesting stories created for the combinations

Mechanical combinations

- Combination test design is mechanical if you can follow an algorithm (or a detailed procedure) to generate the tests
  - Combinatorial testing, such as all-singles and all-pairs
  - Random (use a random number generator) selection of combinations
  - High-volume sampling driven by algorithms developed from a variety of research programs

The selection of the test conditions (the variables and their values) can be done mechanically, but you often need human judgment to decide what tests and oracles are of interest.

RISK-BASED COMBINATION

- **Risk-based** combinations are suggested by such factors as:
  - History of failures in the field
  - Reports of combinations that have been difficult for other products
  - Combinations that are common in the marketplace
  - Expectation of specific classes of problem, such as memory overflow
- See the Multivariable Relationships discussion in the Risk-Based Testing lecture.

SCENARIO-BASED COMBINATION

- Here you provide a meaningful combination based on the benefits that an experienced user would expect from a program of this type.
- This is a **scenario-based** combination.
- You are less likely to achieve a level of coverage like “all pairs” with scenario-based combination testing
- We are more likely to use a suite of scenario tests to thoroughly explore something (like a type of benefit expected by the customer) tied to the value of the product.

Review of lecture 6

We focused on mechanical combination testing, especially all-singles and all-pairs.

- Provide an intuitively appealing coverage model
- Efficient for some important tasks
- Appeal to the mathematically inclined

However, these approaches

- Don't provide test design guidance beyond selection of the values of the variables you are specifically studying
- Are not very risk focused and not very focused on real-life uses or motivating uses of the product.

End of
BBST Test Design

(Yippee!)

References

The most recent version of this reference list is at http://kaner.com/?p=100

REFERENCES

The rest of the slides list references. This set is not exhaustive. We are trying to include:

- All the sources that we specifically relied on in creating this course's slides
- For the techniques this course emphasizes, (function testing; testing tours; risk-based testing; scenario testing; specification analysis; domain testing; combinatorial testing), a sample of sources *we learned from*.
- Suggestions readings. When you come back to these slides later, thinking about applying one of the techniques we mentioned, look at the readings we suggest for that technique. We think these are good starting points for learning about the technique.

REFERENCES

- Active reading (see also Specification-based testing and Concept mapping)
  - Adler, M. (1940). How to mark a book. http://academics.keene.edu/tmendham/documents/AdlerMortimerHowToMarkABook_20060802.pdf
  - Adler, M., & Van Doren, C. (1972). How to Read a Book. Touchstone.
  - Beacon Learning Center. (Undated). Just Read Now. http://www.justreadnow.com/strategies/active.htm
  - Active reading (summarizing Bean, J. Engaging Ideas). (Undated). http://titan.iwu.edu/~writcent/Active_Reading.htm
  - Gause, D.C., & Weinberg, G.M. (1989). Exploring Requirements: Quality Before Design. Dorset House.
  - Hurley, W.D. (1989). A Generative Taxonomy of Application Domains Based on Interaction Semantics. Ph.D. Dissertation, George Washington University. http://dl.acm.org/citation.cfm?id=75960 (on generative taxonomies. See also Jha's and Vijayaraghavan's papers in Failure Mode Analysis, below)
  - PLAN: Predict/Locate/Add/Note. (Undated). http://www.somers.k12.ny.us/intranet/reading/PLAN.html
  - MindTools. (Undated). Essential skills for an excellent career. http://www.mindtools.com
  - Penn State University Learning Centers. (Undated). Active Reading. http://istudy.psu.edu/FirstYearModules/Reading/Materials.html
  - Weill, P. (Undated). Reading Strategies for Content Areas: Part 1 After Reading. www.douglasesd.k12.or.us/LinkClick.aspx?fileticket=6FrvrocjBnk%3d&tabid=1573
- All-pairs testing

  See http://www.pairwise.org/ for more references generally and http://www.pairwise.org/tools.asp for a list of tools.
  - Bach, J., & Schroeder, P. (2004). Pairwise Testing: A Best Practice that Isn't. Proceedings of the 22nd Pacific Northwest Software Quality Conference, 180–196. http://www.testingeducation.org/wtst5/PairwisePNSQC2004.pdf. For Bach's tool, see http://www.satisfice.com/tools/pairs.zip
  - Bolton, M. (2007). Pairwise testing. http://www.developsense.com/pairwiseTesting.html
  - Chateauneuf, M. (2000). Covering Arrays. Ph.D. Dissertation (Mathematics). Michigan Technological University.
  - Cohen, D. M., Dalal, S. R., Fredman, M. L., & Patton, G. C. (1997). The AETG system: An approach to testing based on combinatorial design. IEEE Transactions on Software Engineering, 23(7). http://aetgweb.argreenhouse.com/papers/1997-tse.html. For more references, see http://aetgweb.argreenhouse.com/papers.shtml
  - Czerwonka, J. (2008). Pairwise testing in the real world: Practical extensions to test-case scenarios. http://msdn.microsoft.com/en-us/library/cc150619.aspx. Microsoft's PICT tool is at http://download.microsoft.com/download/f/5/5/f55484df-8494-48fa-8dbd-8c6f76cc014b/pict33.msi

REFERENCES

- All-pairs testing (continued)
  - Jorgensen, P. (2008, 3rd Ed.). Software Testing: A Craftsman's Approach. Auerbach Publications.
  - Kuhn, D. R. & Okun, V. (2006). Pseudo-exhaustive testing for software. 30th Annual IEEE/NASA Software Engineering Workshop. http://csrc.nist.gov/acts/PID258305.pdf
  - Zimmerer, P. (2004). Combinatorial testing experiences, tools, and solutions. International Conference on Software Testing, Analysis & Review (STAR West). http://www.stickyminds.com
- Alpha testing
  - See references on tests by programmers of their own code, or on relatively early testing by development groups. For a good overview from the viewpoint of the test group, see Schultz, C.P., Bryant, R., & Langdell, T. (2005). Game Testing All in One. Thomson Press
- Ambiguity analysis (See also specification-based testing)
  - Bender, R. (Undated). *The Ambiguity Review Process*, http://benderrbt.com/Ambiguityprocess.pdf
  - Berry, D.M., Kamisties, E., & Krieger, M.M. (2003) From contract drafting to software specification: Linguistic sources of ambiguity. http://se.uwaterloo.ca/~dberry/handbook/ambiguityHandbook.pdf
  - Fabbrini, F., Fusani, M., Gnesi, S., & Lami, G. (2000) Quality evaluation of software requirements specifications. Thirteenth International Software & Internet .Quality Week. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.98.4333&rep=rep1&type=pdf
  - Spector, C.C. (1997) Saying One Thing, Meaning Another, Eau Claire, WI: Thinking Publications (reprint at www.superduperinc.com/products/view.aspx?pid=tpx12901)
  - Spector, C.C. (2001). As Far As Words Go: Activities for Understanding Ambiguous Language and Humor, Baltimore: Brookes Publishing.
- Best representative testing (See domain testing)

REFERENCES

- Beta testing
  - Bolton, M. (2001). Effective beta testing. http://www.developsense.com/EffectiveBetaTesting.html
  - Fine, M.R. (2002). Beta Testing for Better Software. Wiley.
  - Schultz, C.P., Bryant, R., & Langdell, T. (2005). Game Testing All in One. Thomson Press.
  - Spolsky, J. (2004). Top twelve tips for running a beta test. http://www.joelonsoftware.com/articles/BetaTest.html
  - Wilson, R. (2005). Expert user validation testing. Unpublished manuscript.
- Boundary testing (See domain testing)
- Bug bashes
  - Berkun, S. (2008). How to run a bug bash. http://www.scottberkun.com/blog/2008/how-to-run-a-bug-bash/
  - Powell, C. (2009). Bug bash. http://blog.abakas.com/2009/01/bug-bash.html
  - Schroeder, P.J. & Rothe, D. (2007). Lessons learned at the stomp. Conference of the Association for Software Testing, http://www.associationforsoftwaretesting.org/?dl_name=Lessons_Learned_at_the_Stomp.pdf
- Build verification
  - Guckenheimer, S. & Perez, J. (2006). Software Engineering with Microsoft Visual Studio Team System. Addison Wesley.
  - Page, A., Johnston, K., & Rollison, B.J. (2009). How We Test Software at Microsoft. Microsoft Press.
  - Raj, S. (2009). Maximize your investment in automation tools. Software Testing Analysis & Review. http://www.stickyminds.com

REFERENCES

- Calculations
  - **Note**: There is a significant, relevant field: Numerical Analysis. The list here merely points you to a few sources I have personally found helpful, not necessarily to the top references in the field.
  - Arsham, H. (2010) Solving system of linear equations with application to matrix inversion. http://home.ubalt.edu/ntsbarsh/business-stat/otherapplets/SysEq.htm
  - Boisvert, R.F., Pozo, R., Remington, K., Barrett , R.F., & Dongarra, J.J. (1997) Matrix Market: A web resource for test matrix collections. In Boisvert, R.F. (1997) (Ed.) Quality of Numerical Software: Assessment and Enhancement. Chapman & Hall.
  - Einarsson, B. (2005). Accuracy and Reliability in Scientific Computing. Society for Industrial and Applied Mathematics (SIAM).
  - Gregory, R.T. & Karney, D.L. (1969). A Collection of Matrices for Testing Computational Algorithms. Wiley.
  - Kaw, A.K. (2008), Introduction to Matrix Algebra. Available from htt p://www.lulu.com/browse/preview.php?fCID=2143570. Chapter 9, Adequacy of Solutions, http://numericalmethods.eng.usf.edu/mws/gen/04sle/mws_gen_sle_spe_adequacy.pdf
- Combinatorial testing. See All-Pairs Testing
- Concept mapping
  - Hyerle, D.N. (2008, 2nd Ed.). Visual Tools for Transforming Information into Knowledge, Corwin.
  - Margulies, N., & Maal, N. (2001, 2nd Ed.) Mapping Inner Space: Learning and Teaching Visual Mapping. Corwin.
  - McMillan, D. (2010). Tales from the trenches: Lean test case design. http://www.bettertesting.co.uk/content/?p=253
  - McMillan, D. (2011). Mind Mapping 101. http://www.bettertesting.co.uk/content/?p=956
  - Moon, B.M., Hoffman, R.R., Novak, J.D., & Canas, A.J. (Eds., 2011). Applied Concept Mapping: Capturing, Analyzing, and Organizing Knowledge. CRC Press.
  - Nast, J. (2006). Idea Mapping: How to Access Your Hidden Brain Power, Learn Faster, Remember More, and Achieve Success in Business. Wiley.
  - Sabourin, R. (2006). X marks the test case: Using mind maps for software design. Better Software. http://www.stickyminds.com/BetterSoftware/magazine.asp?fn=cifea&id=90

REFERENCES

Concept mapping tools:

- http://www.graphic.org/
- http://www.innovationtools.com/resources/mindmapping.asp
- http://www.inspiration.com
- http://www.mindjet.com
- http://www.mindtools.com/mindmaps.html
- http://www.novamind.com
- http://users.edte.utwente.nl/lanzing/cm_bibli.htm
- http://www.xmind.net
- http://en.wikipedia.org/wiki/List_of_concept_mapping_and_mind_mapping_software

- Configuration coverage
  - Black, R. (2002, 2nd Ed.). Managing the Testing Process. Wiley.
  - Kaner, C. (1996). Software negligence and testing coverage. Software Testing, Analysis & Review Conference (STAR). http://www.kaner.com/pdfs/negligence_and_testing_coverage.pdf
  - Pawson, M. (2001). The test matrix. Software Testing & Quality Engineering, 3(6). http://www.stickyminds.com/getfile.asp?ot=XML&id=5005&fn=Smzr1XDD3339filelistfilename1%2Epdf
- Configuration / compatibility testing
  - Frye, C. (2009). Configuration testing: QA pros discuss 10 things you may not know. http://searchsoftwarequality.techtarget.com/news/1361051/Configuration-testing-QA-pros-discuss-10-things-you-may-not-know
  - Kaner, C., Falk, J., & Nguyen, H.Q. (2nd Edition, 2000). Testing Computer Software. Wiley.
  - McCaffrey, J., & Despe, P. (2008). Configuration testing with virtual server, part 2. MSDN Magazine. December. http://msdn.microsoft.com/en-us/magazine/dd252952.aspx
  - Patton, Ron. (2006, 2nd Ed.). Software Testing. SAMS.

REFERENCES

- Constraint checks
  - See our notes in BBST Foundation's presentation of Hoffman's collection of oracles.
  - Hoffman, D. (1999). Heuristic test oracles. Software Testing & Quality Engineering, 1(2), 29-32. http://www.softwarequalitymethods.com/Papers/STQE%20Heuristic.pdf
- Constraints
  - Jorgensen, A.A. (1999). Software Design Based on Operational Modes. Doctoral Dissertation, Florida Institute of Technology. https://cs.fit.edu/Projects/tech_reports/cs-2002-10.pdf
  - Whittaker, J.A. (2002) How to Break Software, Addison Wesley.
- Diagnostics-based testing
  - Al-Yami, A.M. (1996). Fault-Oriented Automated Test Data Generation. Ph.D. Dissertation, Illinois Institute of Technology.
  - Kaner, C., Bond, W.P., & McGee, P.(2004). High volume test automation. Keynote address: *International Conference on Software Testing Analysis & Review* (STAR East 2004). Orlando. http://www.kaner.com/pdfs/HVAT_STAR.pdf (The Telenova and Mentsville cases are both examples of diagnostics-based testing.)
- Domain testing
  - Abramowitz & Stegun (1964), Handbook of Mathematical Functions. http://people.math.sfu.ca/~cbm/aands/frameindex.htm
  - Beizer, B. (1990). Software Testing Techniques (2nd Ed.). Van Nostrand Reinhold.
  - Beizer, B. (1995). Black-Box Testing. Wiley.
  - Binder, R. (2000). Testing Object-Oriented Systems: Addison-Wesley.
  - Black, R. (2009). Using domain analysis for testing. Quality Matters, Q3, 16-20. http://www.rbcs-us.com/images/documents/quality-matters-q3-2009-rb-article.pdf
  - Copeland, L. (2004). A Practitioner's Guide to Software Test Design. Artech House.
  - Clarke, L.A. (1976). A system to generate test data and symbolically execute programs. IEEE Transactions on Software Engineering, 2, 208-215.
  - Clarke, L. A. Hassel, J., & Richardson, D. J. (1982). A close look at domain testing. IEEE Transactions on Software Engineering, 2, 380-390.
  - Craig, R. D., & Jaskiel, S. P. (2002). Systematic Software Testing. Artech House.
  - Hamlet, D. & Taylor, R. (1990). Partition testing does not inspire confidence. IEEE Transactions on Software Engineering, 16(12), 1402-1411.

REFERENCES

- Domain testing (continued)
  - Hayes, J.H. (1999). Input Validation Testing: A System-Level, Early Lifecycle Technique. Ph.D. Dissertation (Computer Science), George Mason University.
  - Howden, W. E. (1980). Functional testing and design abstractions. Journal of Systems & Software, 1, 307-313.
  - Jeng, B. & Weyuker, E.J. (1994). A simplified domain-testing strategy. ACM Transactions on Software Engineering, 3(3), 254-270.
  - Jorgensen, P. C. (2008). Software Testing: A Craftsman's Approach (3rd ed.). Taylor & Francis.
  - Kaner, C. (2004a). Teaching domain testing: A status report. Paper presented at the Conference on Software Engineering Education & Training. http://www.kaner.com/pdfs/teaching_sw_testing.pdf
  - Kaner, C., Padmanabhan, S., & Hoffman, D. (2012) Domain Testing: A Workbook, in preparation.
  - Myers, G. J. (1979). The Art of Software Testing. Wiley.
  - Ostrand, T. J., & Balcer, M. J. (1988). The category-partition method for specifying and generating functional tests. Communications of the ACM, 31(6), 676-686.
  - Padmanabhan, S. (2004). Domain Testing: Divide and Conquer. M.Sc. Thesis, Florida Institute of Technology. http://www.testingeducation.org/a/DTD&C.pdf
  - Schroeder, P.J. (2001). Black-box test reduction using input-output analysis. Ph.D. Dissertation (Computer Science). Illinois Institute of Technology.
  - Weyuker, E. J., & Jeng, B. (1991). Analyzing partition testing strategies. IEEE Transactions on Software Engineering, 17(7), 703-711.
  - Weyuker, E.J., & Ostrand, T.J. (1980). Theories of program testing and the application of revealing subdomains. IEEE Transactions on Software Engineering, 6(3), 236-245.
  - White, L. J., Cohen, E.I., & Zeil, S.J. (1981). A domain strategy for computer program testing. In Chandrasekaran, B., & Radicchi, S. (Ed.), Computer Program Testing (pp. 103-112). North Holland Publishing.
  - http://www.wikipedia.org/wiki/Stratified_sampling
- Dumb monkey testing
  - Arnold, T. (1998), Visual Test 6. Wiley.
  - Nyman, N. (1998). Application testing with dumb monkeys. International Conference on Software Testing Analysis & Review (STAR West).
  - Nyman, N. (2000), Using monkey test tools. Software Testing & Quality Engineering, 2(1), 18-20
  - Nyman, N. (2004). In defense of monkey testing. http://www.softtest.org/sigs/material/nnyman2.htm

REFERENCES

- Eating your own dogfood
  - Page, A., Johnston, K., & Rollison, B.J. (2009). How We Test Software at Microsoft. Microsoft Press.
- Equivalence class analysis (see Domain testing)
- Experimental design
  - Popper, K.R. (2002, 2nd Ed.). Conjectures and Refutations: The Growth of Scientific Knowledge. Routledge.
  - Shadish, W.R., Cook, T.D., & Campbell, D.T. (2002). Experimental and Quasi-Experimental Designs for Generalized Causal Inference, 2nd Ed. Wadsworth.
- Exploratory testing
  - Bach, J. (1999). General functionality and stability test procedure. http://www.satisfice.com/tools/procedure.pdf
  - Bach, J. (2000). Session-based test management. Software Testing & Quality Engineering. http://www.satisfice.com/articles/sbtm.pdf
  - Bach, J. (2007). Exploratory testing explained. In Ryber,. T. (2007). Essential Software Test Design. Fearless Consulting. http://www.satisfice.com/articles/et-article.pdf
  - Bach, J., Bach, J. & Bolton, M. (2009). Exploratory testing dynamics. (v 2.2). http://www.satisfice.com/blog/wp-content/updloads/2009/10/et-dynamics22.pdf
  - Bolton, M. (2011). Resources on Exploratory Testing, Metrics and Other Stuff. http://www.developsense.com/resources.html#exploratory
  - Cox, R., Duisters, P. & van der Laar, J. (2011). Testing in the medical domain. Testing Experience (March), 6-8. http://www.improveqs.nl/files/testingexperience13_03_11_Cox_Duisers_Laar.pdf
  - Hendrickson, E. (2011). Exploratory Testing in an Agile Context. http://www.agilistry.com/downloads/ETinAgile-agile2011-final.pdf
  - Kaner, C. (2006). Exploratory testing after 23 years. Conference of the Association for Software Testing. http://www.kaner.com/pdfs/ETat23.pdf
  - Kaner, C. & Hoffman, D. (2010). Introduction to exploratory test automation. http://kaner.com/pdfs/VISTACONexploratoryTestAutmation.pdf
  - Kohl, J. (2007). Getting started with exploratory testing --Parts 1-4. http://www.kohl.ca/blog/archives/000185.html
  - Kohl, J. (2007). Getting started with exploratory testing --Part 2. http://www.kohl.ca/blog/archives/000186.html

REFERENCES

- Exploratory testing (continued)
  - Kohl, J. (2007). Getting started with exploratory testing --Part 3. http://www.kohl.ca/blog/archives/000187.html
  - Kohl, J. (2007). Getting started with exploratory testing --Part 4. http://www.kohl.ca/blog/archives/000188.htmlRobinson, H. (2010). Exploratory test automation. Conference of the Association for Software Testing. http://www.harryrobinson.net/ExploratoryTestAutomation-CAST.pdf
  - Kohl, J. (2007). Man and machine: Combining the power of the human mind with automation tools. Better Software, December, 20-25. http://www.kohl.ca/articles/ManandMachine_BetterSoftware_Dec2007.pdf
  - Kohl, J. (2011). Documenting exploratory testing. Better Software, May/June, 22-25. http://www.nxtbook.com/nxtbooks/sqe/bettersoftware_0511/#/23/OnePage
- Failure mode analysis: see also Guidewords and Risk-Based Testing.
  - Cheit, R.E. (1990). Setting Safety Standards: Regulation in the Public and Private Sectors. University of California Press. http://ark.cdlib.org/ark:/13030/ft8f59p27j/
  - Department of Defense (1980). Procedures for Performing a Failure Mode, Effects and Criticality Analysis: MIL-STD-1629A. http://sre.org/pubs/Mil-Std-1629A.pdf
  - Department of Defense Patient Safety Center (2004). Failure mode and effects analysis (FMEA): An advisor's guide. AF Patient Safety Program. http://www.fmeainfocentre.com/handbooks/FMEA_Guide_V1.pdf
  - FMEA Info Center (undated). FMEA Info Center: Everything You Want to Know About Failure Mode and Effect Analysis. http://www.fmeainfocentre.com/index.htm (this is a large collection of resources on FMEA).
  - FMEA-FMECA.COM (undated). FMEA Examples. http://fmea-fmeca.com/fmea-examples.html
  - Goddard, P.L. (2000). Software FMEA techniques. Proceedings of the Reliability and Maintainability Symposium, 118-123.
  - Hurley, W.D. (1989). A Generative Taxonomy of Application Domains Based on Interaction Semantics. Ph.D. Dissertation, George Washington University. http://dl.acm.org/citation.cfm?id=75960
  - Jha, A. (2007). A Risk Catalog for Mobile Applications. (Master's Thesis in Software Engineering) Department of Computer Sciences at Florida Institute of Technology. http://www.testingeducation.org/articles/AjayJha_Thesis.pdf
  - Kaner, C., Falk, J., & Nguyen, H.Q. (2nd Edition, 2000b). Bug Taxonomy (Appendix) in Testing Computer Software. Wiley. http://www.logigear.com/logi_media_dir/Documents/whitepapers/Common_Software_Errors.pdf
  - Pentti, H. & Atte, H. (2002). Failure mode and effects analysis of software-based automation systems. http://www.fmeainfocentre.com/handbooks/softwarefmea.pdf
  - SoftRel (undated) Software FMEA service. http://www.softrel.com/fmea.htm

REFERENCES

- Failure mode analysis (continued) (see also Guidewords and Risk-Based Testing):
  - Vijayaraghavan, G. (2002). A Taxonomy of E-Commerce Risks and Failures. (Master's Thesis) Department of Computer Sciences at Florida Institute of Technology. http://www.testingeducation.org/a/tecrf.pdf
  - Vijayaraghavan, G., & Kaner, C. (2002). Bugs in your shopping cart: A taxonomy. 15th International Software Quality Conference (Quality Week). San Francisco, CA. (Best Paper Award.) http://www.testingeducation.org/a/bsct.pdf
  - Vijayaraghavan, G., & Kaner, C.(2003). Bug taxonomies: Use them to generate better tests. Software Testing, Analysis & Review Conference (Star East). Orlando, FL. (Best Paper Award). http://www.testingeducation.org/a/bugtax.pdf
- Feature integration testing
  - Overbaugh, J. (2007). How to do integration testing. http://searchsoftwarequality.techtarget.com/answer/How-to-do-integration-testing
  - Van Tongeren, T. (2001). Functional integration test planning. http://www.stickyminds.com/getfile.asp?ot=XML&id=3693&fn=XUS2004669file1%2Epdf
- Function testing
  - Bolton, M. (2006). The factors of function testing. Better Software. http://www.developsense.com/articles/2006-07-TheFactorsOfFunctionTesting.pdf
  - Craig, R.D., & Jaskiel, S.P. (2002). Systematic Software Testing. See Chapter 5, Test Design (the discussion of inventories). Artech House.
- Function equivalence testing
  - Hoffman, D. (2003). Exhausting your test options. Software Testing & Quality Engineering, 5(4), 10-11
  - Kaner, C., Falk, J., & Nguyen, H.Q. (2nd Edition, 2000). Testing Computer Software. Wiley.
- Functional testing below the GUI
  - Marick, B. (2002). Bypassing the GUI. Software Testing & Quality Engineering, Sept/Oct. 41-47. http://www.exampler.com/testing-com/writings/bypassing-the-gui.pdf
- Guerilla testing
  - Kaner, C., Falk, J., & Nguyen, H.Q. (2nd Edition, 2000). Testing Computer Software. Wiley.
- Guidewords
  - Bach, J. (2006). Heuristic test strategy model, Version 4.8. http://www.satisfice.com/tools/satisfice-tsm-4p.pdf
  - Broomfield, E.J. & Chung, P.W.H. (1994). Hazard identification in programmable systems: A methodology and case study. ACM SIGAPP Applied Computing Review, 2(1), 7-14.

REFERENCES

- Guidewords (continued)
  - Falla, M. (Ed.) (1997). Advances in Safety Critical Systems: Results and Achievements from the DTI/EPSRC R&D Programme in Safety Critical Systems. Chapter 3: Hazard Analysis. http://www.comp.lancs.ac.uk/computing/resources/scs/
  - Fenelon, P. & Hebbron, B. (1994). Applying HAZOP to software engineering models. Risk Management and Critical Protective Systems: Proceedings of SARSS 1994. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.6073&rep=rep1&type=pdf
  - Fenelon, P., McDermid, J.A., Nicholson, M. & Pumfrey, D.J. (1994). Towards integrated safety analysis and design. ACM SIGAPP Applied Computing Review, 2(1), 21-32. http://www-users.cs.york.ac.uk/~djp/publications/djp-acm.pdf
  - Fields, R. Paterno, F., Santoro, C. & Tahmassebi, S. (1999). Comparing design options for allocating communication media in cooperative safety-critical contexts: A method and a case study. ACM Transactions on Computer-Human Interaction, 6(4), 370–398.
  - Gagnat, A.K. & Jansen, V. (2006). Incorporating human factors and interface design into safety-critical systems development. M.Sc. Thesis, Østfold University College, Faculty of Computer Science, http://www.hiof.no/neted/upload/attachment/site/group12/Ann_Katrin_Gagnat_&_Vegar_Jansen_Incorporating_human_factors_and_interface_design_into_safety_critical_systems_development.pdf
  - HAZOP Guidelines (2008). Hazardous Industry Planning Advisory Paper No. 8, NSW Government Department of Planning. http://www.planning.nsw.gov.au/plansforaction/pdf/hazards/haz_hipap8_rev2008.pdf
  - Hewitt, R. (2005). Information-based risk assessment software architecture. Proceedings of the 2005 IEEE Engineering Management Conference, 574-578.
  - Lutz, R. & Nikora, A. (2005, November). Failure Assessment. 1st Int'l Forum on Integrated System Health Engineering and Management for Aerospace (ISHEM'05). http://www.cs.iastate.edu/~rlutz/publications/ishem05.pdf
  - Reinhardt, D. (2006). Certification criteria for emulation technology in the Australian Defence Force Military Avionics Context. 11th Australian Workshop on Safety Related Programmable Systems (SCS'06). Conferences in Research and Practice in Information Technology. 69, 79-92. http://crpit.com/confpapers/CRPITV69Reinhardt.pdf
  - Stone, G.R. (2005). On arguing the safety of large systems. 10th Australian Workshop on Safety related Programmable Systems. Conferences in Research and Practice in Information Technology, 55, 69-75. http://crpit.com/confpapers/CRPITV55Stone.pdf
  - Ye, F. & Kelly, T. (1994). Contract-based justification for COTS component within safety-critical applications. 9th Australian Workshop on Safety Related Programmable Systems (SCS' 04). Conferences in Research & Practice in Information Technology, 47, 13-22. http://crpit.com/confpapers/CRPITV47Ye.pdf

REFERENCES

- Installation testing
  - Agruss, C. (2000). Software installation testing: How to automate tests for smooth system installation. Software Testing & Quality Engineering, 2 (4). http://www.stickyminds.com/getfile.asp?ot=XML&id=5001&fn=Smzr1XDD1806filelistfilename1%2Epdf
  - Bach, J. (1999), Heuristic risk-based testing, Software Testing & Quality Engineering, 1 (6), 22-29. http://www.satisfice.com/articles/hrbt.pdf
  - Kumar, P. (2010). Installation testing – why and how? Testing Circus, 1(2), 7-11. http://testingcircus.com/OctoberNovember2010.aspx
  - Noggle, B.J. (2011). Testing the installer. The Testing Planet. March, http://wiki.softwaretestingclub.com/w/file/fetch/39449474/TheTestingPlanet-Issue4-March2011.pdf
  - Shinde, V. (2007). Software installation/uninstallation testing. http://www.softwaretestinghelp.com/software-installationuninstallation-testing
- Interoperability testing
  - European Telecommunications Standards Institute (undated). Interoperability test specification. http://portal.etsi.org/mbs/testing/interop/interop.htm
  - Simulations Interoperability Standards Organization (2010). Commercial Off-the-Shelf (COTS) Simulation Package Interoperability (CSPI) Reference Mode. http://www.sisostds.org/DigitalLibrary.aspx?Command=Core_Download&EntryId=30829
- Load testing
  - Asbock, S. (2000). Load Testing for eConfidence. Segue.
  - Barber, S. (2006). Remember yesterday. Software Test & Performance Magazine. January, 42-43. http://www.perftestplus.com/resources/016PeakPerf.pdf
  - Savoia, A. (2000). The science and art of web site load testing. International Conference on Software Testing Analysis & Review (STAR East), Orlando. www.stickyminds.com/getfile.asp?ot=XML&id=1939&fn=XDD1939filelistfilename1.pdf
  - Savoia, A. (2001). Three web load testing blunders and how to avoid them. Software Testing & Quality Engineering, 3(3), 54-59. http://www.stickyminds.com/s.asp?F=S5034_MAGAZINE_2

REFERENCES

- Localization testing
  - Bolton, M. (2006, April). Where in the world? Better Software. http://www.developsense.com/articles/2006-04-WhereInTheWorld.pdf
  - Chandler, H.M. & Deming, S.O (2nd Ed. in press). The Game Localization Handbook. Jones & Bartlett Learning.
  - Ratzmann, M., & De Young, C. (2003). Galileo Computing: Software Testing and Internationalization. Lemoine International and the Localization Industry Standards Association. http://www.automation.org.uk/downloads/documentation/galileo_computing-software_testing.pdf
  - Savourel, Y. (2001). XML Internationalization and Localization. Sams Press.
  - Singh, N. & Pereira, A. (2005). The Culturally Customized Web Site: Customizing Web Sites for the Global Marketplace. Butterworth-Heinemann.
  - Smith-Ferrier, G. (2006). .NET Internationalization: The Developer's Guide to Building Global Windows and Web Applications. Addison-Wesley Professional.
  - Uren, E., Howard, R. & Perinotti, T. (1993). Software Internationalization and Localization. Van Nostrand Reinhold.
- Logical expression testing
  - Amman, P., & Offutt, J. (2008). Introduction to Software Testing. Cambridge University Press.
  - Beizer, B. (1990). Software Testing Techniques (2nd Ed.). Van Nostrand Reinhold.
  - Copeland, L. (2004). A Practitioner's Guide to Software Test Design. Artech House (see Chapter 5 on decision tables).
  - Jorgensen, P. (2008, 3rd Ed.). Software Testing: A Craftsman's Approach. Auerbach Publications (see Chapter 7 on decision tables).
  - Brian Marick (2000) modeled testing of logical expressions by considering common mistakes in designing/coding a series of related decisions. Testing for Programmers. http://www.exampler.com/testing-com/writings/half-day-programmer.pdf.
  - MULTI. Marick implemented his approach to testing logical expressions in a program, MULTI. Tim Coulter and his colleagues extended MULTI and published it (with Marick's permission) at http://sourceforge.net/projects/multi/

REFERENCES

- Long-sequence testing
  - Claessen, K. (undated). QuickCheck 2.4.0.1: Automatic testing of Haskell programs. http://hackage.haskell.org/cgi-bin/hackage-scripts/package/QuickCheck
  - Koopman, P. (undated). The Ballista ® Project: COTS software robustness testing. http://www.cs.cmu.edu/afs/cs/project/edrc-ballista/www/
  - McGee, P. & Kaner, C. (2004). Experiments with high volume test automation." Workshop on Empirical Research in Software Testing, International Symposium on Software Testing and Analysis. http://www.kaner.com/pdfs/MentsvillePM-CK.pdf
- Mathematical oracle
  - See our notes in BBST Foundation's presentation of Hoffman's collection of oracles.
  - Abramowitz & Stegun (1964), Handbook of Mathematical Functions. http://people.math.sfu.ca/~cbm/aands/frameindex.htm
  - Hoffman, D. (1999). Heuristic test oracles. Software Testing & Quality Engineering, 1(2), 29-32. http://www.softwarequalitymethods.com/Papers/STQE%20Heuristic.pdf
- Numerical analysis (see Calculations)
- Paired testing
  - Kaner, C. & Bach, J. (2001). Exploratory testing in pairs. Software Testing Analysis & Review (STAR West). http://www.kaner.com/pdfs/exptest.pdf
  - Kohl, J. (2004). Pair testing: How I brought developers into the test lab. Better Software, 6 (January), 14-16. http://www.kohl.ca/articles/pairtesting.html
  - Lambert, R. (2009). Pair testing. http://www.softwaretestingclub.com/forum/topics/pair-testing
- Pairwise testing (see All-Pairs testing)
- Performance testing
  - Barber, S. (2010) PerfTestPlus website, http://www.perftestplus.com/pubs.htm
  - du Plessis, J. (2009). Resource monitoring during performance testing. http://www.stickyminds.com/getfile.asp?ot=XML&id=14959&fn=XUS231346159file1%2Epdf
  - Meier, J.D., Farre, C., Bansode, P., Barber, S., & Rea, D. (2007). Performance Testing Guidance for Web Applications. Redmond: Microsoft Press.

REFERENCES

- Programming or software design
  - Roberts, E. (2005, 20th Ed.). Thinking Recursively with Java. Wiley.
- Psychological considerations
  - Bendor, J. (2005). The perfect is the enemy of the best: Adaptive versus optimal organizational reliability. Journal of Theoretical Politics. 17(1), 5-39.
  - Rohlman, D.S. (1992). The Role of Problem Representation and Expertise in Hypothesis Testing: A Software Testing Analogue. Ph.D. Dissertation, Bowling Green State University.
  - Teasley, B.E., Leventhal, L.M., Mynatt, C.R., & Rohlman, D.S. (1994). Why software testing is sometimes ineffective: Two applied studies of positive test strategy. Journal of Applied Psychology, 79(1), 142-155.
  - Whittaker, J.A. (2000). What is software testing? And why is it so hard? IEEE Software, Jan-Feb. 70-79.
- Quicktests
  - Andrews, M., & Whittaker, J.A. (2006), How to Break Web Software, Addison Wesley.
  - Hendrickson, E. (2006), Test heuristics cheat sheet. http://testobsessed.com/wp-content/uploads/2011/04/testheuristicscheatsheetv1.pdf
  - Hunter, M. J. (2010). You are not done yet. http://www.thebraidytester.com/downloads/YouAreNotDoneYet.pdf
  - Jorgensen, A.A. (1999). Software Design Based on Operational Modes. Doctoral Dissertation, Florida Institute of Technology. https://cs.fit.edu/Projects/tech_reports/cs-2002-10.pdf
  - Jorgensen, A.A. (2003). Testing with hostile data streams. ACM SIGSOFT Software Engineering Notes, 28(2). http://cs.fit.edu/media/TechnicalReports/cs-2003-03.pdf
  - Kaner, C. (1988), Testing Computer Software, McGraw-Hill, presented a list of 480 common software problems. http://logigear.com/articles-by-logigear-staff/445-common-software-errors.html
  - Kaner, C. & Johnson, B. (1999) Styles of exploration, 7th Los Altos Workshop on Software Testing. http://www.kaner.com/pdfs/LAWST7StylesOfExploration.pdf
  - Nguyen, H.Q., Johnson, B., & Hackett, M. (2003, 2nd ed), Testing Applications on the Web. Wiley.
  - Whittaker, J.A. (2002) How to Break Software, Addison Wesley.
  - Whittaker, J.A. & Thompson, H.H. (2004). How to Break Software Security. Addison Wesley.

REFERENCES

- Random testing
  - Ciupa, I., Leitner, A., Oriol, M., & Meyer, B. (2007). Experimental Assessment of Random Testing for Object-Oriented Software. International Symposium on Software Testing and Analysis. http://staff.unak.is/not/andy/MScTesting0708/Lectures/ExpRandomTestingOOEiffel.pdf
  - Hamlet, D. (2006). When only random testing will do. Proceedings of the 1st International Workshop on Random Testing. http://web.cecs.pdx.edu/~hamlet/rt.pdf
  - Hamlet, D. (2002) Random testing. Encyclopedia of Software Engineering. http://web.cecs.pdx.edu/~hamlet/random.pdf
  - Robinson, H. (2004). Things that find bugs in the night. http://www.stickyminds.com/s.asp?F=S7331_COL_2
- Regression testing
  - Bach, J. (1999). Test automation snake oil. http://www.satisfice.com/articles/test_automation_snake_oil.pdf
  - Buwalda, H. (undated). Key success factors for keyword-driven testing. http://logigear.com/articles-by-logigear-staff/389--key-success-factors-for-keyword-driven-testing.html
  - Engstrom, E., Skoglund, M. & Runeson, P. (2008). Empirical evaluations of regression test selection techniques: A systematic review. Conference on Empirical Software Engineering and Measurement. 22-31. http://portal.acm.org/citation.cfm?doid=1414004.1414011
  - Groder, C. (1999). Building maintainable GUI tests. In Fewster, M. & Graham, D. (1999). Software Test Automation. Addison-Wesley
  - Harrold, M.J. & Orso, A. (2008). Retesting software during development and maintenance. Frontiers of Software Maintenance. http://pleuma.cc.gatech.edu/aristotle/pdffiles/harrold_orso_fosm08.pdf
  - Hoffman, D. (2007). Avoiding the "test and test again" syndrome. Conference of the Association for Software Testing. http://www.softwarequalitymethods.com/H-Papers.html#TestTest2
  - Kaner, C. (1997). Improving the maintainability of automated test suites. Software QA, (4)(4). http://www.kaner.com/pdfs/autosqa.pdf
  - Kaner, C. (1998). Avoiding shelfware: A manager's view of automated GUI testing." Software Testing Analysis & Review (STAR East). http://www.kaner.com/pdfs/shelfwar.pdf
  - Kaner, C. (2009). The value of checklists and the danger of scripts: What legal training suggests for testers. Conference of the Association for Software Testing. http://www.kaner.com/pdfs/ValueOfChecklists.pdf

REFERENCES

- Regression testing (continued)
  - Leung, H.K.N. & White, L.J. (1989(. Insights into regression testing. Proceedings of the International Conference on Software Maintenance. 60-69.
  - Marick, B.M. (1999). When should a test be automated? International Conference on Software Testing Analysis and Review (STAR East). http://www.exampler.com/testing-com/writings/automate.pdf
  - Marick, B.M. (2005). Working your way out of the automated GUI testing tarpit (parts 1, 2, 3). http://www.testingreflections.com/node/view/3059
  - Marick, B.M. (undated). How many bugs do regression tests find? http://www.sqa.fyicenter.com/art/How_Many_Bugs_Do_Regression_Tests_Find.html
  - Memon, A.M. & Xie, Q. (2004) Empirical evaluation of the fault-detection effectiveness of smoke regression test cases for GUI-based software. Proceedings of the International Conference on Software Maintenance. http://www.cs.umd.edu/~atif/papers/MemonICSM2004.pdf
  - Mugridge, R. & Cunningham. W. (2005) Fit for Developing Software: Framework for Integrated Tests. Prentice Hall.
  - Nguyen, H.Q., Hackett, M., & Whitlock, B.K. (2006). Happy About Global Test Automation. Happy About books.
  - Onoma, A.K., Tsai, W.T., Poonawala, M.H., & Sugunama, H. (1998). Regression testing in an industrial environment. Communications of the ACM. 41(5), 81-86
  - Pettichord, B. (2001a). Seven steps to test automation success. http://www.io.com/~wazmo/papers/seven_steps.html
  - Pettichord, B. (2001b). Success with test automation. http://www.io.com/~wazmo/succpap.htm
  - Rothermel, G. & Harrold, M. (1997). Experience with regression test selection. Empirical Software Engineering, 2(2), 178-188. http://www.cc.gatech.edu/aristotle/Publications/Papers/wess96.ps
- Requirements-based testing
  - Bach, J. (1999). Risk and requirements-based testing. IEEE Computer, June, 113-114. http://www.satisfice.com/articles/requirements_based_testing.pdf
  - Bender, R. (2009). Requirements Based Testing Process Overview. http://benderrbt.com/Bender-Requirements%20Based%20Testing%20Process%20Overview.pdf
  - Culbertson, R., Brown, C., & Cobb, G. (2002). Rapid Testing. Prentice Hall.

REFERENCES

- Requirements-based testing (continued)
  - Whalen, M.W., Rajan, A., Heimdahl, M.P.E., & Miller, S.P.)2006). Coverage metrics for requirements-based testing. Proceedings of the 2006 International Symposium on Software Testing and Analysis. http://portal.acm.org/citation.cfm?id=1146242
  - Wiegers, K.E. (1999). Software Requirements. Microsoft Press.
- Risk-based testing
  - Bach, J. (1999). Heuristic risk-based testing. Software Testing & Quality Engineering. http://www.satisfice.com/articles/hrbt.pdf
  - Bach, J. (2000a). Heuristic test planning: Context model. http://www.satisfice.com/tools/satisfice-cm.pdf
  - Bach, J. (2000b). SQA for new technology projects. http://www.satisfice.com/articles/sqafnt.pdf
  - Bach, J. (2003). Troubleshooting risk-based testing. Software Testing & Quality Engineering, May/June, 28-32. http://www.satisfice.com/articles/rbt-trouble.pdf
  - Becker, S.A. & Berkemeyer, A. (1999). The application of a software testing technique to uncover data errors in a database system. Proceedings of the 20th Annual Pacific Northwest Software Quality Conference, 173-183.
  - Berkovich, Y. (2000). Software quality prediction using case-based reasoning. M.Sc. Thesis (Computer Science). Florida Atlantic University.
  - Bernstein, P.L. (1998). Against the Gods: The Remarkable Story of Risk. Wiley.
  - Black, R. (2007). Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional. Wiley.
  - Clemen, R.T. (1996, 2nd ed.) Making Hard Decisions: An Introduction to Decision Analysis. Cengage Learning.
  - Copeland, L. (2004). A Practitioner's Guide to Software Test Design. Artech House.
  - DeMarco, T. & Lister, T. (2003). Waltzing with Bears. Managing Risk on Software Projects. Dorset House.
  - Dorner, D. (1997). The Logic of Failure. Basic Books.
  - Gerrard, P. & Thompson, N. (2002). Risk-Based E-Business Testing. Artech House.
  - HAZOP Guidelines (2008). Hazardous Industry Planning Advisory Paper No. 8, NSW Government Department of Planning. http://www.planning.nsw.gov.au/plansforaction/pdf/hazards/haz_hipap8_rev2008.pdf
  - Hillson, D. & Murray-Webster, R. (2007, 2nd Ed.). Understanding and Managing Risk Attitude. Gower. http://www.risk-attitude.com
  - Hubbard, D.W. (2009). The Failure of Risk Management: Why It's Broken and How to Fix It. Wiley.
  - Jorgensen, A.A. (2003). Testing with hostile data streams. ACM SIGSOFT Software Engineering Notes, 28(2). http://cs.fit.edu/media/TechnicalReports/cs-2003-03.pdf

REFERENCES

- Risk-based testing (continued)
  - Jorgensen, A.A. & Tilley, S.R. (2003). On the security risks of not adopting hostile data stream testing techniques. 3rd International Workshop on Adoption-Centric Software Engineering (ACSE 2003), p. 99-103. http://www.sei.cmu.edu/reports/03sr004.pdf
  - Kaner, C. (2008). Improve the power of your tests with risk-based test design. Quality Assurance Institute QUEST conference. http://www.kaner.com/pdfs/QAIriskKeynote2008.pdf
  - Kaner, C., Falk, J., & Nguyen, H.Q. (2nd Edition, 2000a). Testing Computer Software. Wiley.
  - Neumann, P.G. (undated). The Risks Digest: Forum on Risks to the Public in Computers and Related Systems. http://catless.ncl.ac.uk/risks
  - Perrow, C. (1999). Normal Accidents: Living with High-Risk Technologies. Princeton University Press (but read this in conjunction with Robert Hedges' review of the book on Amazon.com).
  - Petroski, H. (1992). To Engineer is Human: The Role of Failure in Successful Design. Vintage.
  - Petroski, H. (2004). Small Things Considered: Why There is No Perfect Design. Vintage.
  - Petroski, H. (2008). Success Through Failure: The Paradox of Design. Princeton University Press.
  - Pettichord, B. (2001). The role of information in risk-based testing. International Conference on Software Testing Analysis & Review (STAR East). http://www.stickyminds.com
  - Reason, J. T. (1997). Managing the Risks of Organizational Accident. Ashgate Publishing.
  - Schultz, C.P., Bryant, R., & Langdell, T. (2005). Game Testing All in One. Thomson Press (discussion of defect triggers).
  - Software Engineering Institute's collection of papers on project management, with extensive discussion of project risks. https://seir.sei.cmu.edu/seir/
  - Weinberg, G. (1993). Quality Software Management. Volume 2: First Order Measurement. Dorset House.
- Rounding errors (see Calculations)
- Scenario testing (See also Use-case-based testing)
  - Anggreeni, I., & van der Voort, M. (2007) Tracing the scenarios in scenario-based product design: A study to support scenario generation. Technical Report TR-CTIT-07-70, Centre for Telematics and Information Technology, University of Twente, Enschede. ISSN 1381-3625. http://eprints.eemcs.utwente.nl/11231/01/TR-CTIT-07-70.pdf
  - Alexander, D. (2000). Scenario methodology for teaching principles of emergency management. Disaster Prevention & Management, Vol. 9(2), pp. 89-97.

REFERENCES

- Scenario testing (continued)
  - Alexander, I., & Maiden, N. (2004). Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle. Wiley.
  - Bolton, M. (2007). Users we don't like. Developsense.com. http://www.developsense.com/articles/2007-08-UsersWeDontLike.pdf
  - Bolton, M. (2007). Why we do scenario testing. Developsense.com. http://www.developsense.com/blog/2010/05/why-we-do-scenario-testing/
  - Buwalda, H. (2000a). The three holy grails of test development. presented at EuroSTAR conference.
  - Buwalda, H. (2000b). Soap Opera Testing. International Software Quality Week Europe conference, Brussels. http://www.logigear.com/resource-center/software-testing-articles-by-logigear-staff/246-soap-opera-testing.html
  - Buwalda, H. (2007a). Key principles of test design. http://www.logigear.com/newsletter-2007/304-key-principles-of-test-design.html
  - Buwalda, H. (2007b). The first holy grail of test design. http://www.logigear.com/newsletter-2007/305-the-first-holy-grail-of-test-design.html
  - Buwalda, H. (2007c). The second holy grail of test design. http://www.logigear.com/newsletter-2007/306-the-second-holy-grail-of-test-design.html
  - Buwalda H. (2007d). The third holy grail of test design. http://www.logigear.com/newsletter-2007/307-the-third-holy-grail-of-test-design.html
  - Buwalda, H. (2008). The potential and risks of keyword based testing. http://logigear.com/newsletter-2008/342-the-potential-and-risks-of-keyword-based-testing.html
  - Buwalda, H. (undated). Key success factors for keyword-driven testing. http://logigear.com/articles-by-logigear-staff/389--key-success-factors-for-keyword-driven-testing.html
  - Buwalda, H., Janssen, D. & Pinkster, I. (2002). Integrated Test Design and Automation: Using the TestFrame Method. Addison-Wesley.
  - Carroll, J.M. (ed.) (1995). Scenario-Based Design. Wiley.
  - Carroll, J.M. (1995). Development, Scenario-Based Design: Envisioning Work and Technology in System. Wiley.
  - Carroll, J.M. (1999). Five reasons for scenario-based design. Proceedings of the 32nd Hawaii International Conference on System Sciences, http://www.massey.ac.nz/~hryu/157.757/Scenario.pdf
  - Guckenheimer, S. & Perez, J. (2006). Software Engineering with Microsoft Visual Studio Team System. Addison Wesley.
  - Heijden, Kes van der (1996). Scenarios: The Art of Strategic Conversation. Wiley.

REFERENCES

- Scenario testing (continued) (See also Use-case-based testing and Task analysis)
  - Kahn, H. (1967). The use of scenarios. In Kahn, Herman & Wiener, Anthony (1967). The Year 2000: A Framework for Speculation on the Next Thirty-Three Years, pp. 262-264. http://www.hudson.org/index.cfm?fuseaction=publication_details&id=2214
  - Kaner, C. (2003). An introduction to scenario testing. http://www.kaner.com/pdfs/ScenarioIntroVer4.pdf
  - Memon, A.M., Pollack, M.E., & Soffa, M.L. (2000). A planning-based approach to GUI testing. International Software Quality Week, San Francisco. www.cs.umd.edu/~atif/papers/MemonSQW2000.pdf; http://www.cs.umd.edu/~atif/presentations/SQW2000.pdf
  - Ringland, G. (1998). Scenario Planning: Managing for the Future. Wiley.
  - Rippel, M. & Teply, P. (2009). Operational Risk -- Scenario Analysis. Working Papers IES 2008/15, Charles University Prague, Faculty of Social Sciences, Institute of Economic Studies, revised Sep 2008. http://ideas.repec.org/p/fau/wpaper/wp2008_15.html
  - Rosson, M.B., & Carroll, J.M. (2002). Usability Engineering: Morgan Kaufmann.
  - Rothman, J., & Lawrence, B. (1999). Testing in the dark. Software Quality & Design Engineering, 1(2). Pp. 34-39 http://www.jrothman.com/Papers/Pragmaticstrategies.html
  - Wack, P. (1985a). Scenarios: Uncharted waters ahead. Harvard Business Review 63(5), 74-89. http://tuvalu.santafe.edu/events/workshops/images/d/d9/Wack.pdf
  - Wack, P. (1985b). Scenarios: Shooting the rapids. Harvard Business Review 63(6), 139-150. http://www.scribd.com/doc/4489875/Wack-Shooting-the-rapids
  - Walker, W.E. (1994). The use of scenarios and gaming in crisis management planning and training. Presented at the conference, The Use of Scenarios for Crisis Management, Netherlands Ministry of Home Affairs, at the Netherlands Institute for Fire Service & Disaster Mgmt, Arnhem, November,16-18.
  - Weber, B. (2008) Information Commerce 1997 – Scenario Mapping Changes Beliefs. http://www.strategykinetics.com/scenario_planning/
- Self-verifying data
  - Nyman, N. (1999). Self-verifying data: Testing without an oracle. International Conference on Software Testing Analysis & Research (STAR East), Orlando. http://www.stickyminds.com/getfile.asp?ot=XML&id=2918&fn=XDD2918filelistfilename1%2Epdf
  - Knaus, R. Aougab, H., & Bentahar, N. (2004). Software Reliability: A Preliminary Handbook. McLean, VA: United States Department of Transportation: Federal Highway Administration (see the discussions of wrapping, especially Chapter 6). http://www.fhwa.dot.gov/publications/research/safety/04080/04080.pdf
  - Romero, G. & Gauvin, C. (1998). Self-verifying communications testing. United States Patent US7409618, http://www.patents.com/self-verifying-communications-testing-7409618.html

REFERENCES

- Specification-based testing (See also active reading; See also ambiguity analysis)
  - Bach, J. (1999). Reframing requirements analysis. IEEE Computer, 32(2), 120-122. http://www.satisfice.com/articles/reframing_requirements.pdf
  - Bach, J. (2006). Heuristic test strategy model, Version 4.8. http://www.satisfice.com/tools/satisfice-tsm-4p.pdf
  - Black, R. (2007). Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional. Wiley.
  - Donat, M. R. (1998). A Discipline of Specification-Based Test Derivation, Ph.D. Dissertation (Computer Science). University of British Columbia. http://www.nlc-bnc.ca/obj/s4/f2/dsk2/ftp02/NQ34519.pdf
  - Hayes, J.H. (1999). Input Validation Testing: A System-Level, Early Lifecycle Technique. Ph.D. Dissertation (Computer Science), George Mason University.
  - Jacky, J., Veanes, M., Campbell, C. & Schulte, W. (2008). Model-Based Software Testing and Analysis with C#. Cambridge University Press.
  - Kaner, C. (1998). Liability for product incompatibility. Software QA, 5(4), p. 33ff. http://www.kaner.com/pdfs/liability.pdf
  - Kaner, C. (2003). Liability for defective documentation. Conference of the ACM SIGDOC. http://www.kaner.com/pdfs/liability_sigdoc.pdf
  - Lawrence, B. (undated). http://www.coyotevalley.com/tools.htm
- State-model-based testing
  - Auer, A.J. (1997). State Testing of Embedded Software. Ph.D. Dissertation (Computer Science). Oulun Yliopisto (Finland).
  - Becker, S.A. & Whittaker, J.A. (1997). Cleanroom Software Engineering Practices. IDEA Group Publishing.
  - Buwalda, H. (2003). Action figures. Software Testing & Quality Engineering. March/April 42-27. http://www.logigear.com/articles-by-logigear-staff/245-action-figures.html
  - El-Far, I. K. (1999), Automated Construction of Software Behavior Models, Masters Thesis, Florida Institute of Technology, 1999.
  - El-Far, I. K. & Whittaker, J.A. (2001), Model-based software testing, in Marciniak, J.J. (2001). Encyclopedia of Software Engineering, Wiley. http://testoptimal.com/ref/Model-based Software Testing.pdf
  - Jorgensen, A.A. (1999). Software Design Based on Operational Modes. Doctoral Dissertation, Florida Institute of Technology. https://cs.fit.edu/Projects/tech_reports/cs-2002-10.pdf

REFERENCES

- State-model-based testing (continued)
  - Katara, M., Kervinen, A., Maunumaa, M., Paakkonen, T., & Jaaskelainen, A. (2007). Can I have some model-based GUI tests please? Providing a model-based testing service through a web interface. Conference of the Association for Software Testing. http://practise.cs.tut.fi/files/publications/TEMA/cast07-final.pdf
  - Mallery, C.J. (2005). On the Feasibility of Using FSM Approaches to Test Large Web Applications. M.Sc. Thesis (EECS). Washington State University.
  - Page, A., Johnston, K., & Rollison, B.J. (2009). How We Test Software at Microsoft. Microsoft Press.
  - Robinson, H. (1999a). Finite state model-based testing on a shoestring. http://www.stickyminds.com/getfile.asp?ot=XML&id=2156&fn=XDD2156filelistfilename1%2Epdf
  - Robinson, H. (1999b). Graph theory techniques in model-based testing. International Conference on Testing Computer Software. http://sqa.fyicenter.com/art/Graph-Theory-Techniques-in-Model-Based-Testing.html
  - Robinson, H. Model-Based Testing Home Page. http://www.geocities.com/model_based_testing/
  - Rosaria, S., & Robinson, H. (2000). Applying models in your testing process. Information & Software Technology, 42(12), 815-24. http://www.harryrobinson.net/ApplyingModels.pdf
  - Schultz, C.P., Bryant, R., & Langdell, T. (2005). Game Testing All in One. Thomson Press.
  - Utting, M., & Legeard, B. (2007). Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann.
  - Vagoun, T. (1994). State-Based Software Testing. Ph.D. Dissertation (Computer Science). University of Maryland College Park.
  - Whittaker, J.A. (1992). Markov Chain Techniques for Software Testing and Reliability Analysis. Ph.D. Dissertation (Computer Science). University of Tennessee.
  - Whittaker, J.A. (1997). Stochastic software testing. Annals of Software Engineering, 4, 115-131.
- Stress testing
  - http://en.wikipedia.org/wiki/Stress_testing
  - Beizer, B. (1984). Software System Testing and Quality Assurance. Van Nostrand. See also: http://www.faqs.org/faqs/software-eng/testing-faq/section-15.html
- Task analysis (see also Scenario testing and Use-case-based testing)
  - Crandall, B., Klein, G., & Hoffman, R.B. (2006). Working Minds: A Practitioner's Guide to Cognitive Task Analysis. MIT Press.
  - Draper, D. & Stanton, N. (2004). The Handbook of Task Analysis for Human-Computer Interaction. Lawrence Erlbaum.

REFERENCES

- Task analysis (continued) (see also Scenario testing and Use-case-based testing)
  - Ericsson, K.A. & Simon, H.A. (1993). Protocol Analysis: Verbal Reports as Data (Revised Edition). MIT Press.
  - Gause, D.C., & Weinberg, G.M. (1989). Exploring Requirements: Quality Before Design. Dorset House.
  - Hackos, J.T. & Redish, J.C. (1998). User and Task Analysis for Interface Design. Wiley.
  - Jonassen, D.H., Tessmer, M., & Hannum, W.H. (1999). Task Analysis Methods for Instructional Design.
  - Robertson, S. & Robertson, J. C. (2006, 2nd Ed.). Mastering the Requirements Process. Addison-Wesley Professional.
  - Schraagen, J.M., Chipman, S.F., & Shalin, V.I. (2000). Cognitive Task Analysis. Lawrence Erlbaum.
  - Shepard, A. (2001). Hierarchical Task Analysis. Taylor & Francis.
- Test design / test techniques (in general)
  - Bach, J. (2000). Heuristic Test Planning: Context Model. http://www.satisfice.com/tools/satisfice-cm.pdf
  - Bach, J. (2006). Heuristic test strategy model. Version 4.8. http://www.satisfice.com/tools/satisfice-tsm-4p.pdf
  - Black, R. (2007). Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional. Wiley.
  - Buwalda, H. (2007). Key principles of test design. http://logigear.com/newsletter-2007/304-key-principles-of-test-design.html
  - Copeland, L. (2004). A Practitioner's Guide to Software Test Design. Artech House.
  - Collard, R. (2001). Test Design Fundamentals. International Conference on Software Testing Analysis & Review. (STAR West)
  - Desikan, S. & Gopalaswamy, R. (2006). Software Testing: Principles and Practices. Pearson Education.
  - Edgren, R. (2010). Review of properties in Kaner's What is a Good Test Case? http://thetesteye.com/blog/2010/05/review-of-properties-in-kaners-what-is-a-good-test-case
  - Edgren, R. (2011). The Little Black Book on Test Design. http://thetesteye.com/blog/2011/09/the-little-black-book-on-test-design/
  - Emilsson, H., Jansson, M., Edgren, R. (2010). Software Quality Characteristics 1.0. http://thetesteye.com/posters/TheTestEye_SoftwareQualityCharacteristics.pdf
  - Jorgensen, P. (2008, 3rd Ed.). Software Testing: A Craftsman's Approach. Auerbach Publications.
  - Kaner, C. (2003). What is a good test case? http://www.kaner.com/pdfs/GoodTest.pdf
  - Kaner, C. (2004b). The ongoing revolution in software testing. Software Test & Performance conference. http://www.kaner.com/pdfs/TheOngoingRevolution.pdf

REFERENCES

- Test design / test techniques (in general)
  - Kaner, C., Bach, J., & Pettichord, B. (2001). Lessons Learned in Software Testing: Chapter 3: Test Techniques. http://media.techtarget.com/searchSoftwareQuality/downloads/Lessons_Learned_in_SW_testingCh3.pdf
  - Kelly, M.D. (2007). Specialists and other myths: Because you aren't a specialist doesn't mean you can't do it. Conference of the Association for Software Testing. www.michaeldkelly.com/pdfs/CAST2007_SpecialistsAndOtherMyths.pdf
  - Kelly, M.D. (2009). Software testing: Assessing risk and scope. http://searchsoftwarequality.techtarget.com/podcast/Software-Testing-Assessing-risk-and-scope
  - Loveland, S., Miller, G. Prewitt, R., & Shannon, M. (2005). Software Testing Techniques: Finding the Techniques that Matter. Charles River Media.
  - McMillan, D. (2010). Tales from the trenches: Lean test case design. http://www.bettertesting.co.uk/content/?p=253
  - Nguyen, H.Q., Johnson, B., & Hackett, M. (2003). Testing Applications on the Web, 2nd Ed. Wiley.
  - Page, A., Johnston, K., & Rollison, B.J. (2009). How We Test Software at Microsoft. Microsoft Press.
  - Perry, W.E. (2006). Effective Methods for Software Testing. Wiley.
  - Rajani, R., & Oak, P. (2004). Software Testing: Effective Methods, Tools & Techniques. Tata McGraw-Hill.
  - Ryber,. T. (2007). Essential Software Test Design. Fearless Consulting.
  - Schultz, C.P., Bryant, R., & Langdell, T. (2005). Game Testing All in One. Thomson Press.
  - Sutton, M., Greene, A., & Amini, P. (2007). Fuzzing: Brute Force Vulnerability Discovery. Addison Wesley.
  - Takanen, A., DeMott, J., & Miller, C. (2008). Fuzzing for Software Security Testing and Quality Assurance. Artech House.
  - Whittaker, J.A. (2002). How to Break Software. Addison Wesley.
- Test idea catalogs
  - Edgren, R. (2009). More and better test ideas. EuroSTAR. http://www.thetesteye.com/papers/redgren_moreandbettertestideas.pdf
  - Edgren, R. (2011). The Little Black Book on Test Design. http://thetesteye.com/blog/2011/09/the-little-black-book-on-test-design/
  - Hendrickson, E. (2006). Test Heuristics Cheat Sheet. http://testobsessed.com/wp-content/uploads/2011/04/testheuristicscheatsheetv1.pdf
  - Hunter, M. J. (2010). You are not done yet. http://www.thebraidytester.com/downloads/YouAreNotDoneYet.pdf
  - Kaner, C., Padmanabhan, S., & Hoffman, D. (2012) Domain Testing: A Workbook, in press.
  - Marick, B.M. (1994). The Craft of Software Testing: Subsystems Testing Including Object-Based and Object-Oriented Testing. Prentice-Hall. Updated catalog: http://www.exampler.com/testing-com/writings/catalog.pdf

REFERENCES

- Test idea catalogs (continued)
  - Marick, B.M. (undated). A short catalog of test ideas for… http://www.exampler.com/testing-com/writings/short-catalog.pdf
  - Nguyen, H.Q., Johnson, B., & Hackett, M. (2003, 2nd ed), Testing Applications on the Web. Wiley (Appendices D through H).
  - Sabourin, R. (2008). Ten things you might not know about sources of testing ideas. Better Software, April. http://www.amibugshare.com/articles/Article_10_Sources_of_testing_ideas.pdf
- Testing skill

  Many of the references in this collection are about the development of testing skill. However, a few papers stand out, to me, as exemplars of papers that focus on activities or structures designed to help testers improve their day-to-day testing skills. We need more of these.
  - Bach, J. (2006). Heuristic test strategy model, Version 4.8. http://www.satisfice.com/tools/satisfice-tsm-4p.pdf
  - Gärtner, M. (2011) Testing Dojos: Craftsmanship in Software Testing: Blog at http://www.testingdojo.org/tiki-index.php. Expanded from Gaertner, M. (2010, Winter). Testing dojos: Another way to gain software testing experience. Methods & Tools. http://www.methodsandtools.com/mt/download.php?winter10
  - Hendrickson, E. (2006), Test heuristics cheat sheet. http://testobsessed.com/wp-content/uploads/2011/04/testheuristicscheatsheetv1.pdf
  - Hunter, M. J. (2010). You are not done yet. http://www.thebraidytester.com/downloads/YouAreNotDoneYet.pdf
- Tours
  - Bolton, M. (2009). Of testing tours and dashboards. http://www.developsense.com/blog/2009/04/of-testing-tours-and-dashboards
  - Craig, R. D., & Jaskiel, S. P. (2002). Systematic Software Testing. Artech House. (see their discussions of inventories)
  - Goucher, A. (2009). 'Exploratory Software Testing' – a cheat of a book. http://adam.goucher.ca/?p=1225
  - Kelly, M.D. (2005). Touring Heuristic. http://www.michaeldkelly.com/archives/50
  - Kelly, M.D. (2006). Taking a tour through test country. A guide to tours to take on your next test project. Software Test and Performance Magazine, February, 20-25. http://michaeldkelly.com/pdfs/Taking%20a%20Tour%20Through%20Test%20Country.pdf; http://www.softwaretestpro.com/Item/2752/Taking-a-Tour-Through-Test-Country/Test-and-QA-Video-Exploratory-Strategy-Performance-Functional-Software-Testing-Unit-User-Web
  - Kohl, J. (2006). Modeling test heuristics. http://www.kohl.ca/blog/archives/000179.html
  - Laplante, P. (2009). Exploratory testing for mission critical, real-time, and embedded systems. Part of the IEEE Reliability Society 2009 Annual Technology Report. http://paris.utdallas.edu/IEEE-RS-ATR/document/2009/2009-08.pdf
  - Whittaker, J.A. (2009). Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design. Addison-Wesley.

REFERENCES

- Usability testing
  - Cooper, A. (2004). The Inmates are Running the Asylum: Why High Tech Products Drive Us Crazy and How to Restore the Sanity. Pearson Education.
  - Cooper, A., Reimann, R. & Cronin, D. (2007). About Face 3: The Essentials of Interaction Design. Wiley.
  - Dumas, J.S. & Loring, B.A. (2008). Moderating Usability Tests: Principles and Practices for Interacting. Morgan Kaufmann.
  - Fiedler, R.L., & Kaner, C. (2009). "Putting the context in context-driven testing (an application of Cultural Historical Activity Theory)." Conference of the Association for Software Testing. http://www.kaner.com/pdfs/FiedlerKanerCast2009.pdf
  - Ives, B., Olson, M.H., & Baroudi, J.J. (1983). The measurement of user information systems. Communications of the ACM, 26(10), 785-793. http://portal.acm.org/citation.cfm?id=358430
  - Krug, S. (2005, 2nd Ed.). Don't Make Me Think: A Common Sense Approach to Web Usability. New Riders Press.
  - Kuniavsky, M. (2003). Observing the User Experience: A Practitioner's Guide to User Research. Morgan Kaufmann.
  - Lazar, J., Fend, J.H., & Hochheiser, H. (2010). Research Methods in Human-Computer Interaction. Wiley.
  - Nielsen, J. (1994). Guerrilla HCI: Using discount usability engineering to penetrate the intimidation barrier. http://www.useit.com/papers/guerrilla_hci.html
  - Nielsen, J. (1999). Designing Web Usability. Peachpit Press.
  - Nielson, J. & Loranger, H. (2006). Prioritize Web Usability. MIT Press.
  - Norman, D.A. (2010). Living with Complexity. MIT Press.
  - Norman, D.A. (1994). Things that Make Us Smart: Defending Human Attributes in the Age of the Machine. Basic Books.
  - Norman, D.A. & Draper, S.W. (1986). User Centered System Design: New Perspectives on Human-Computer Interaction. CRC Press.
  - Patel, M. & Loring, B. (2001). Handling awkward usability testing situations. Proceedings of the Human Factors and Ergonomics Society 45th Annual Meeting. 1772-1776.

REFERENCES

- Usability testing (continued)
  - Platt, D.S. (2006). Why Software Sucks. Addison-Wesley.
  - Rubin, J., Chisnell, D. & Spool, J. (2008). Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests. Wiley.
  - Smilowitz, E.D., Darnell, M.J., & Benson, A.E. (1993). Are we overlooking some usability testing methods? A comparison of lab, beta, and forum tests. Proceedings of the Human Factors and Ergonomics Society 37th Annual Meeting, 300-303.
  - Stone, D., Jarrett, C., Woodroffe, M. & Minocha, S. (2005). User Interface Design and Evaluation. Morgan Kaufmann.
  - Tullis, T. & Albert, W. (2008). Measuring the User Experience: Collecting, Analyzing, and Presenting Usability Metrics (Interactive Technologies). Morgan Kaufmann.
- Use-case based testing (see also Scenario testing and Task analysis)
  - Adolph, S. & Bramble, P. (2003). Patterns for Effective Use Cases. Addison-Wesley.
  - Alexander, Ian & Maiden, Neil. Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle.
  - Alsumait, A. (2004). User Interface Requirements Engineering: A scenario-based framework. Ph.D. dissertation (Computer Science), Concordia University.
  - Berger, Bernie (2001) "The dangers of use cases employed as test cases," STAR West conference, San Jose, CA. www.testassured.com/docs/Dangers.htm
  - Charles, F.A. (2009). Modeling scenarios using data. STP Magazine. http://www.quality-intelligence.com/articles/Modelling%20Scenarios%20Using%20Data_Paper_Fiona%20Charles_CAST%202009_Final.pdf
  - Cockburn, A.(2001). Writing Effective Use Cases. Addison-Wesley.
  - Cohn, M. (2004). User Stories Applied: For Agile Software Development. Pearson Education.
  - Collard, R. (July/August 1999). Test design: Developing test cases from use cases. Software Testing & Quality Engineering, 31-36.
  - Hsia, P., Samuel, J. Gao, J. Kung, D., Toyoshima, Y. & Chen, C. (1994). Formal approach to scenario analysis. IEEE Software, 11(2), 33-41.
  - Jacobson, I. (1995). The use-case construct in object-oriented software engineering. In John Carroll (ed.) (1995). Scenario-Based Design. Wiley.
  - Jacobson, I., Booch, G. & Rumbaugh, J. (1999). The Unified Software Development Process. Addison-Wesley.
  - Jacobson, I. & Bylund, S. (2000) The Road to the Unified Software Development Process. Cambridge University Press.
  - Kim, Y. C. (2000). A Use Case Approach to Test Plan Generation During Design. Ph.D. Dissertation (Computer Science). Illinois Institute of Technology.
  - Kruchten, P. (2003, 3rd Ed.). The Rational Unified Process: An Introduction. Addison-Wesley.

REFERENCES

- Use-case based testing (continued) (see also Scenario testing and Task analysis)
  - Samuel, J. (1994). Scenario analysis in requirements elicitation and software testing. M.Sc. Thesis (Computer Science), University of Texas at Arlington.
  - Utting, M., & Legeard, B. (2007). Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann.
  - Van der Poll, J.A., Kotze, P., Seffah, A., Radhakrishnan, T., & Alsumait, A. (2003). Combining UCMs and formal methods for representing and checking the validity of scenarios as user requirements. Proceedings of the South African Institute of Computer Scientists and Information Technologists on Enablement Through Technology. http://dl.acm.org/citation.cfm?id=954014.954021
  - Zielczynski, P. (2006). Traceability from use cases to test cases. http://www.ibm.com/developerworks/rational/library/04/r-3217/
- User interface testing
  - Adzic, G. (2007). Effective user interface testing. http://gojko.net/2007/09/25/effective-user-interface-testing/
  - Apple Computer (1993). Macintosh Human Interface Guidelines. Apple.
  - Apple Computer (2011). Apple Human Interface Guidelines: User Experience. http://developer.apple.com/library/mac/#documentation/UserExperience/Conceptual/AppleHIGuidelines
  - Dudziak, T. (2005). How to unit test the user interface of web applications. ApacheCon US. http://floyd.openqa.org/HowToUnitTestTheUserInterfaceOfWebApplications.pdf
  - McKay, E.N. (1999). Developing User Interfaces for Microsoft Windows. Microsoft Press.
  - Microsoft (2010). Windows User Experience Guidelines for Windows 7 and Windows Vista. http://msdn.microsoft.com/en-us/library/aa511258.aspx
  - Nielsen, J. (2001). Coordinating User Interfaces for Consistency. Morgan Kaufmann.
  - Olson, D.R. (1998). Developing User Interfaces. Morgan Kaufmann.
  - Olson, D.R. (2009). Building Interactive Systems: Principles for Human-Computer Interaction. Course Technology.
  - Visual Studio 2010 (undated). Testing the user interface with automated UI tests. http://msdn.microsoft.com/en-us/library/dd286726%28VS.100%29.aspx
- User testing (see beta testing)
  - Albert, W., Tullis, T. & Tedesco, D. (2010). Beyond the Usability Lab: Conducting Large-Scale Online User Experience Studies. Morgan Kaufmann.
  - Wang, E., & Caldwell, B. (2002). An empirical study of usability testing: Heuristic evaluation vs. user testing. Proceedings of the Human Factors and Ergonomics Society 46th Annual Meeting. 774-778.