

# *Black Box Software Testing*

## *(Professional Seminar)*

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

Professor of Computer Sciences  
Florida Institute of Technology

### **Section:13**

## **Specification-Based Testing**

Summer, 2002

Contact Information:

kaner@kaner.com

[www.kaner.com](http://www.kaner.com) (testing website)

[www.badsoftware.com](http://www.badsoftware.com) (legal website)

I grant permission to make digital or hard copies of this work for personal or classroom use, with or without fee, provided that (a) copies are not made or distributed for profit or commercial advantage, (b) copies bear this notice and full citation on the first page, and if you distribute the work in portions, the notice and citation must appear on the first page of each portion, (c) each page bear the notice "Copyright (c) Cem Kaner" or if you changed the page, "Adapted from Notes Provided by Cem Kaner". Abstracting with credit is permitted. The proper citation for this work is Cem Kaner, *A Course in Black Box Software Testing (Professional Version)*, Summer-2002, [www.testing-education.org](http://www.testing-education.org). To copy otherwise, to republish or post on servers, or to distribute to lists requires prior specific permission and a fee. Request permission to republish from [kaner@kaner.com](mailto:kaner@kaner.com).

# *Specification-Driven Testing*

## **Tag line:**

- “Verify every claim.”

## **Fundamental question or goal**

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

## **Paradigmatic case(s)**

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

# *Specification-Driven Testing*

## **Strengths**

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

## **Blind spots**

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

# *Traceability Matrix*

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

# *Traceability Matrix*

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

**The rows are test cases.**

**The cells show which test case tests which items.**

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

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

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

# *Specification*

## **Tasks**

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

# *Specification*

## **Skills**

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

# *Specification*

## **Skills**

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

# *Example: Requirements Questions*

**Important to trace from requirements to implications**

## **Exercise**

- Give a list of questions
  - » Examples are the test documentation requirements questions (in the test documentation section) and the automation maintainability questions at ([www.kaner.com/pdfs/shelfwar.pdf](http://www.kaner.com/pdfs/shelfwar.pdf))
- For each question
  - » Ask the students to name at least two decisions that they would make on the basis of the answer to the question
- Make this a small group exercise, splitting up the questions, groups fill flipcharts, then bring back to full class.

# *Breaking Statements into Elements*

**Make / read a statement about the program**

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

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

# *Breaking Statements into Elements: An Example*

*Quality is value to some person*

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

## *Reviewing a Specification for Completeness*

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

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

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

# *Reviewing a Specification for Completeness*

## **Spec Review using the Satisfice Model, continued**

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

# *Getting Information*

## *When There Is No Specification*

*(Suggestions from some brainstorming sessions.)*

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

# *Getting Information*

## *When There Is No Specification*

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

# *Sample Exam Questions*

Describe a traceability matrix.

- How would you build a traceability matrix for the TI Interactive product?
- What is the traceability matrix used for?
- What are the advantages and risks associated with driving your testing using a traceability matrix?
- Give examples of advantages and risks that you would expect to deal with if you used a traceability matrix for the TI Interactive product. Answer this in terms of two of the main features of the TI Interactive product. You can choose which two features.

