

# *Black Box Software Testing*

## *(Professional Seminar)*

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

Professor of Computer Sciences  
Florida Institute of Technology

### **Section:1**

### **Opening Exercise - Triangle**

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.

# About Cem Kaner

**I'm in the business of improving software customer satisfaction.**

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

## Current employment

- Professor of Software Engineering, Florida Institute of Technology
- Private practice in the Law Office of Cem Kaner

## Books

- *Testing Computer Software* (1988; 2nd edition with Hung Nguyen and Jack Falk, 1993). This received the *Award of Excellence* in the Society for Technical Communication's *Northern California Technical Publications Competition* and has the lifetime best sales of any book in the field.
- *Bad Software: What To Do When Software Fails* (with David Pels). Ralph Nader called this book "a how-to book for consumer protection in the Information Age."

## Education

- J.D. (law degree, 1993). Elected to the American Law Institute, 1999.
- Ph.D. (experimental psychology, 1984) (trained in *measurement theory* and in *human factors*, the field concerned with making hardware and software easier and safer for humans to use).
- B.A. (primarily mathematics and philosophy, 1974).
- Certified in Quality Engineering (American Society for Quality, 1992). Examiner (1994, 1995) for the California Quality Awards.
- I also co-founded and/or co-host the Los Altos Workshops on Software Testing, the Software Test Managers' Roundtable, the Austin Workshop on Test Automation, the Workshop on Model-Based Testing, and the Workshop on Heuristic & Exploratory Techniques.

# *Notice*

These course notes are copyrighted.

For uses outside those listed in the license on the first page, please request permission from me at [kaner@kaner.com](mailto:kaner@kaner.com).

These notes were originally developed in co-authorship with Hung Quoc Nguyen. James Bach has contributed substantial material. I also thank Jack Falk, Elizabeth Hendrickson, Doug Hoffman, Bob Johnson, Brian Lawrence, Melora Svoboda, and the participants in the Los Altos Workshops on Software Testing and the Software Test Managers' Roundtables. Additional acknowledgements appear below.

These notes include legal information, but you are not my legal client. I do not provide legal advice in the course. I may use a question of yours as a teaching tool and answer in a way that would "normally" be true but I cannot explore enough details in a classroom to respond with a competent legal opinion. My answer could be completely inappropriate for your particular situation. I cannot accept responsibility for actions that you might take in response to my comments in this course. If you need legal advice, please consult your own attorney.

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.

# *Contents*

<b>1</b>	<b>16</b>	Opening exercise: triangle	<b>15</b>	<b>342</b>	Paradigm: stress
<b>2</b>	<b>25</b>	An example test series	<b>16</b>	<b>346</b>	Paradigm: regression
<b>3</b>	<b>43</b>	Boundaries & equivalence	<b>17</b>	<b>359</b>	Paradigm: exploratory
<b>4</b>	<b>70</b>	Completeness is impossible	<b>18</b>	<b>437</b>	Paradigm: user
<b>5</b>	<b>82</b>	Overview of development	<b>19</b>	<b>441</b>	Paradigm: scenario
<b>6</b>	<b>98</b>	Black box testing group	<b>20</b>	<b>456</b>	Paradigm: stochastic
<b>7</b>	<b>112</b>	Bug advocacy	<b>21</b>	<b>483</b>	Combination testing
<b>8</b>	<b>242</b>	Intro to black box design	<b>22</b>	<b>512</b>	Intro to test documentation
<b>9</b>	<b>252</b>	Testing paradigms overview	<b>23</b>	<b>519</b>	Scripting test cases
<b>10</b>	<b>261</b>	Paradigm: function testing	<b>24</b>	<b>527</b>	Docs requirements analysis
<b>11</b>	<b>265</b>	Paradigm: domain testing	<b>25</b>	<b>545</b>	Questioning strategies
<b>12</b>	<b>281</b>	Reusable test matrices	<b>26</b>	<b>587</b>	Relationship tables
<b>13</b>	<b>293</b>	Paradigm: spec-driven	<b>27</b>	<b>599</b>	Objectives lists
<b>14</b>	<b>310</b>	Paradigm: risk-based	<b>28</b>	<b>603</b>	Managing automation

# *Contents*

<b>29</b>	<b>628</b>	Automation architectures	<b>43</b>		
<b>30</b>	<b>658</b>	User documentation	<b>44</b>		
<b>31</b>	<b>669</b>	Metrics	<b>45</b>		
<b>32</b>	<b>706</b>	Status reporting	<b>46</b>		
<b>33</b>	<b>715</b>	Project planning	<b>47</b>		
<b>34</b>	<b>742</b>	Legal issues	<b>48</b>		
<b>35</b>	<b>754</b>	Career planning for testers	<b>49</b>		
<b>36</b>	<b>789</b>	Recruiting testers	<b>50</b>		
<b>37</b>	<b>810</b>	Learning styles	<b>51</b>		
<b>38</b>	<b>822</b>	Resources on the Net	<b>52</b>		
<b>39</b>			<b>53</b>		
<b>40</b>			<b>54</b>		
<b>41</b>			<b>55</b>		
<b>42</b>			<b>56</b>		

# *About This Course*

This course is an introduction to black box software testing. Black box testing involves testing software from the customer's view, without knowledge of the underlying code. The alternative to black box testing is glass box (design tests on the basis of your knowledge of the code). Both approaches find defects, and there are significant patterns in the kinds of defects that they find.

This course is based on my book (with Jack Falk and Hung Quoc Nguyen), *Testing Computer Software* (2nd edition). These lecture notes update the book. As an update, they provide much more material than we can cover in class.

This class is designed to be customized. From about 10 days worth of material, we'll select a 5-day subset for classroom use. The rest is for you to study in your spare time.

A final note about the intent of the class. Because it is impossible to test everything, you will always have to make tradeoffs. For example, what is the right ratio of time spent documenting tests vs. actually doing testing? I'll try to help you see the factors that questions like this depend on, and I'll suggest specific answers for some specific circumstances. But at work, you'll have to make your own evaluations of your own situation, and come up with your own answer. The course's goal is to help you improve your ability to make judgments like these.

I revise this course frequently. Suggestions for improvement are *always* welcome, and they make a big difference. Thanks for coming, and for your feedback.

# *Demographics: How long have you worked in:*

- software testing

0-3 months \_\_\_\_\_ 3-6 months \_\_\_\_\_ 6 mo-1 year \_\_\_\_\_  
1-2 years \_\_\_\_\_ 2-5 years \_\_\_\_\_ > 5 years \_\_\_\_\_

- programming

- » Any experience \_\_\_\_\_
- » Production programming \_\_\_\_\_

- management

- » Testing group
- » Any management

- marketing

- documentation

- customer care

- traditional QC

# *Themes of the Course*

## **The Themes**

1. Management (not Testing) heads up QA.
2. Many development methods work.
3. Testing is done in context.
4. The program doesn't work. Therefore, your task is to find errors.
5. Complete testing requires a nearly infinite series of tests.
6. Be methodical.
7. Look for powerful representatives of classes of tests.
8. Groups often develop a better range of insights than individuals.
9. Cost-justify your processes.
10. Think in terms of quality costs.
11. Automation is software development.
12. Communication must be effective.
13. Change is normal.
14. Test planning can be evolutionary.

# *Themes of the Course*

## **1. Management (not Testing) heads up QA.**

A product's head of QA is the person who makes the decisions that determine its quality. That happens long before the start of testing. Test a lousy program forever and you'll end up with an expensive, well-tested, lousy program.

Watch out for project managers who tell you that YOU are responsible for the quality of the product (not them). It's a game that can cause you excessive and undeserved stress.

Your task is to find and clearly report software errors. (See *Testing Computer Software*, Chapter 15, and the Appendix paper on *Negotiating Testing Resources*.)

## **2. Many development methods work.**

School training might lead you to believe that there are only a few "proper" ways to manage projects. In practice, there have been many successful approaches. Companies and project managers have their own styles. Some are much less formal than others. Some are quite formal, but aren't in the traditional "waterfall" mold.

Some companies are appallingly disorganized, and some companies' quality standards are terrible (whether their processes are organized or not).

You will certainly want to form your own opinion of the situations that you're in, but take some time in forming your judgments and look carefully at the actual results of your company's efforts. (See *Testing Computer Software*, page 255 onward.)

*(Themes, continued . . .)*

### **3. Testing is done in context**

Rather than looking for The One True Way to run a test group, we should recognize that the testing group is a service organization and that its work is done in a context. Life cycles, testing group missions, testing techniques must fit within the overall organization. We should look at/for relevant factors (perform a requirements analysis) to guide our decisions.

### **4. The program doesn't work. Therefore, your task is to find errors.**

Nobody would pay you to test if their program didn't have bugs. All programs have bugs. Any change to a program can cause new bugs, and any aspect of the program can be broken.

When someone tells you that your task is to “verify that the program is working” then they're telling you that you're going to fail in your task every time you find a bug. If they're serious:

- you must be doing a final release test on files to go onto the master disks; OR
- the person who's telling you this doesn't understand your job; OR
- the programmer/project manager who's telling you this is talking you out of doing your job; OR
- you should talk further with your manager to understand the charter of your group.

*If you set your mind to showing that a program works correctly, you'll be more likely to miss problems than if you want and expect the program to fail.*

*(Themes, continued . . .)*

## **5. Complete testing requires a nearly infinite series of tests.**

There's a nearly infinite number of paths through any non-trivial program, and a virtually infinite set of combinations of data that you can feed the program. You can't test them all. No one can test them all.

Therefore, your task is to find bugs -- not to find *all* the bugs (you won't) and not to verify that the program has no bugs (it does).

Certainly, you want to find as many bugs as possible, you want to find the most serious bugs, and you want to find them as early as possible.

Your challenges will require judgment, tradeoffs, and efficiency.

## **6. Be methodical.**

Black box testing isn't just banging away at the keyboard. To have any hope of doing an efficient job you must be methodical:

- Break the testing project into tasks and subtasks so that you have a good idea of what has to be done. (It's handy to break down work into 4-hour-sized jobs.)
- Track what you've done so you can avoid:
  - » duplicated effort
  - » misunderstood depth and extent of testing
  - » missing areas or testing them late.
- Prioritize the tasks.

*(Themes, continued . . .)*

## **7. Look for powerful representatives of classes of tests.**

Two tests belong to the same *equivalence class* if you expect the same result (pass / fail) of each. An equivalence class might include thousands of test cases. You don't have time to test them all. Instead, pick a few representative members of the class.

*Boundaries* mark a point or zone of transition from one equivalence class to another (such as the largest number you are allowed to enter into a field, and the number that is 1 larger than the largest valid value). This is where the program is most likely to fail, so these are your best selections from an equivalence class.

Some classes have no firm boundaries (imagine compatibility testing, and the class of LaserJet 4-compatible printers). In this case, pick representative cases, but look for worst cases (such as an almost-compatible).

## **8. Groups often develop a better range of insights than individuals.**

Testers working together in pairs often find more bugs and more interesting bugs than they would find on their own. Testers brainstorming as a group generate a wider range of dimensions or risks to investigate. In general, group work can facilitate the development of creative ideas, while keeping the group focused on the task at hand.

Testing in pairs is also effective for training testers who are new or new to the project.

(Themes, continued . . .)

## **9. Cost-justify your processes.**

Test groups are often asked to do several tasks that don't directly help them find and report bugs. Every minute that you spend on such tasks is a minute that you aren't finding bugs, reporting bugs, or finding a more efficient way to find or report bugs.

These tasks might have great value, but you must evaluate them carefully. This includes the detailed documentation of your testing process.

You might not be the person who does the cost/benefit analysis, but you can still suggest ways to improve your efficiency to your manager, and you should realize that reasonable cost/benefit considerations might have led your company to drop paperwork that testing courses or books might have led you to expect.

## **10. Think in terms of quality costs.**

Testing accounts for only *some* of the money that your company spends on quality.

Quality-related costs include:

- Prevention costs: Cost of preventing customer dissatisfaction, including errors or weaknesses in software, design, documentation, and support.
- Appraisal costs: Cost of inspection (testing, reviews, etc.).
- Internal failure costs: Cost of dealing with errors discovered during development and testing.
- External failure costs: Cost of dealing with errors that affect your customers, after the product is released.

Analyzing situations in these terms will give you a broader understanding, and will often help you make your arguments more effectively or find allies for your position.

*(Themes, continued . . .)*

## **11. Automation is software development.**

Test automation is like any form of automation.

Many test groups attempt to automate with minimal planning, inadequate budgets, absurd schedules, no underlying architecture, and no documentation. These efforts fail -- if not immediately, then when the test group tries to reuse the tests and discovers they provide an unknown level of coverage and are unmaintainable.

When you automate, use the development methods you expect of the development teams whose work you test.

## **12. Communication must be effective.**

From your first day on the job, you are writing reports that could end up on the company president's desk, or in court.

Your bug reports advise people of difficult situations -- the problems they report can affect the project schedule, hurt the company's cash flow, get someone fired. The clearer your reports are, the more likely it is that the company will make reasonable business decisions about them.

Computer Science majors are often poorly trained in persuasive writing, technical writing, oral argument, and face-to-face negotiation. These are core skills for your job.

*(Themes, continued . . .)*

### **13. Change is normal.**

Projects' requirements change. The market changes. And we come to understand the product more thoroughly as we build it.

Some companies accept late changes into their products as a matter of course. As a new tester, you might decide quickly that this is a poor way to do business. It might be, and it might not be. Get to know this company and its business before you let yourself get negative.

If your company does change specifications and design as it goes, there is no point standing in front of the train screaming "Stop changing!" You have to take steps to make yourself more effective in dealing with late changes.

### **14. Test planning can be evolutionary.**

Rather than designing all of your test cases and then running your tests, you can develop the test plan gradually, in parallel with the testing effort.

If you write your full plan, then develop all your cases, then test, much of the planning and development might be made obsolete by program design changes before you run a single test on the changed part of the program.

An evolutionary approach can help you save you a lot of wasted time if the product's design changes during the alpha and beta test phases, because your test plan immediately leads to test cases, which are immediately run.

# *Testing Computer Software*

---

## **Part 1.**

### **An Introductory Exercise**

# *Introductory Exercise*

The program reads three integer values from a card. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral.

» From Glen Myers, *The Art of Software Testing*

- *Write a set of test cases that would adequately test this program.*
- *Please write your name on your answer so that we can hand it back to you. Hand it in when you are done.*

# *Introductory Exercise--Notes*

---

# *Introductory Exercise--Notes*

---

# *Introductory Exercise--Notes*

---

# *Notes on the Exercise*

**Several classes of issues were missed by most students. For example:**

- Few students checked whether they were producing valid triangles. (1,2,3) and (1,2,4) cannot be the lengths of any triangle.
  - » *Knowledge of the subject matter of the program under test will enable you to create test cases that are not directly suggested by the specification. If you lack that knowledge, you will miss key tests. (This knowledge is sometimes called “domain knowledge”, not to be confused with “domain testing.”)*
- Few students checked non-numeric values, bad delimiters, or non-integers.
- The only boundaries tested were at MaxInt or 0.

# *Myers' Answer*

- **Test case for a *valid* scalene triangle**
- **Test case for a valid equilateral triangle**
- **Three test cases for valid isosceles triangles (a=b, b=c, a=c)**
- **One, two or three sides has zero value (5 cases)**
- **One side has a negative**
- **Sum of two numbers equals the third (e.g. 1,2,3) is invalid b/c not a triangle (tried with 3 permutations a+b=c, a+c=b, b+c=a)**
- **Sum of two numbers is less than the third (e.g. 1,2,4) (3 permutations)**
- **Non-integer**
- **Wrong number of values (too many, too few)**

*List 10 tests that you'd run that aren't in Myers' list.*

---

- 1.
- 2.
- 3.
- 4.
- 5.
- 6.
- 7.
- 8.
- 9.
- 10.

**(Do or finish this AFTER we complete Part 2)**

