

Black Box Software Testing

(Academic Course -Fall 2001)

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

Florida Institute of Technology

Section: 2:

An Example Test Series

Contact Information:

kaner@kaner.com

www.kaner.com (testing practitioners)

www.badsoftware.com (software law)

www.testingeducation.org (education research)

Copyright (c) Cem Kaner 2001.

I grant permission to make digital or hard copies of this work for personal or classroom use, 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. Abstracting with credit is permitted. The proper citation for this work is Cem Kaner, *A Course in Black Box Software Testing (Academic Version)*, Fall 2001, 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.

Copyright (c) Cem Kaner 2001.

Testing Computer Software

An Example Test Series:

The Program's First Tests

Assigned Reading:

Kaner, *The Impossibility of Complete Testing*, at www.kaner.com

Let's Work an Example: A Program's First Tests

- This is a practical overview of the testing process.
 - It shows non-testers what happens early in black box testing.
 - It shows experienced testers how I prioritize tasks, how I start developing a test plan in parallel with the project plan, and how I report and follow through on bugs.
 - It introduces the idea of boundary analysis as one way to gain efficiency in testing. It also illustrates combinatorial analysis and makes the point that we can estimate the vastness of the testing task. Finally, this section illustrates the value of focused group brainstorming.
- The example is trivially simple. That's deliberate. I stripped this example to the bare bones to let us focus almost exclusively on the testing issues.

Example Test Series

Here is the program's specification:

- *This program is designed to add two numbers, which you will enter*
- *Each number should be one or two digits*
- *The program will print the sum. Press `Enter` after each number*
- *To start the program, type `ADDER`*
- **Before you start testing, do you have any questions about the spec?**
 - » Refer to Testing Computer Software, Chapter 1, page 1

The First Cycles of Testing

Here's my basic strategy for dealing with new code:

- 1 Start with mainstream-level tests. *Test the program with easy-to-pass values that will be taken as serious issues if the program fails.*
- 2 Test broadly, rather than deeply. *Check out all parts of the program quickly before focusing.*
- 3 Look for more powerful tests. *Once the program can survive the easy tests, put on your thinking cap and look systematically for challenges.*
- 4 Pick boundary conditions. *There will be too many good tests. You need a strategy for picking and choosing.*
- 5 Do some exploratory testing. *Run new tests every week, from the first week to the last week of the project.*

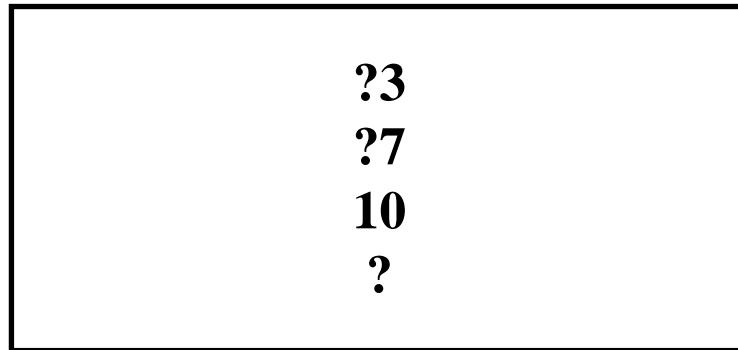
Refer to Testing Computer Software, Chapter 1

1. A Mainstream Level Test

For the first test, try a pair of easy values, such as 3 plus 7.

Here is the screen display that results from that test.

Are there any bug reports that you would file from this?



```
?3
?7
10
?
```

Refer to Testing Computer Software, Chapter 1

A Typical Problem Report Form

PROBLEM REPORT # _____

REPORTER _____

DATE ____/____/____

PROGRAM _____ RELEASE _____ VERSION _____

CONFIGURATION _____

REPORT TYPE ____

1 - Coding error 4 - Documentation

2 - Design Issue 5 - Query

3 - Suggestion

SEVERITY ____

1 - Fatal

2 - Serious

3 - Minor

ATTACHMENTS (Y/N) ____

Description: _____

PROBLEM SUMMARY _____

CAN YOU REPRODUCE THE PROBLEM (Y/N/S/U) _____

PROBLEM AND HOW TO REPRODUCE IT _____

SUGGESTED FIX (*optional*) _____

2. Test Broadly Not Deeply

- The objective of early testing is to flush out the big problems as quickly as possible.
- You will explore the program in more depth as it gets more stable.
- There is no point hammering a design into oblivion if it is going to change. Report as many problems as you think it will take to force a change, and then move on.

3. Look for More Powerful Tests

Brainstorming Rules:

- The goal is to get lots of ideas. You are brainstorming together to discover categories of possible tests.
- There are more great ideas out there than you think.
- Don't criticize others' contributions.
- Jokes are OK, and are often valuable.
- Work *later*, alone or in a much smaller group, to eliminate redundancy, cut bad ideas, and refine and optimize the specific tests.
- Facilitator and recorder keep their opinions to themselves.
We'll work more on brainstorming and, generally, on thinking in groups later.

3. Look for More Powerful Tests

What?

Why?

•	_____	_____
•	_____	_____
•	_____	_____
•	_____	_____
•	_____	_____
•	_____	_____
•	_____	_____
•	_____	_____
•	_____	_____
•	_____	_____
•	_____	_____
•	_____	_____

Refer to Testing Computer Software,
pages 4-5, for examples.

4. *Equivalence Class & Boundary Analysis*

- *There are too many tests.*
- **There are $199 \times 199 = 39,601$ test cases for valid values:**
 - **definitely valid: 0 to 99**
 - **might be valid: -99 to -1**
- **There are infinitely many invalid cases:**
 - **100 and above**
 - **-100 and below**
 - **anything non-numeric**
- **Some people want to automate these tests.**
 - **Can you run all the tests?**
 - **How will you tell whether the program passed or failed?**
- ***We cannot afford to run every possible test. We need a method for choosing a few tests that will represent the rest. Equivalence analysis is the most widely used approach.***
 - refer to Testing Computer Software pages 4-5 and 125-132

4. Classical Equivalence Class and Boundary Analysis

- To avoid unnecessary testing, partition (divide) the range of inputs into groups of equivalent tests.
- Then treat an input value from the equivalence class as representative of the full group.
- 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 good members of equivalence classes to use.

» Myers, Art of Software Testing, 45

I'm leaving the definitions of equivalence and boundary fuzzy for now. We'll come back to them in the next class. Today, we're looking at the basic idea.

4. Traditional Presentation (Myers)

- **One input or output field**
 - The “valid” values for the field fit within one (1) easily specified range.
 - Valid values: -99 to 99
 - Invalid values
 - < -99
 - > 99

4. Myers' Boundary Table

Variable	Valid Case Equivalence Classes	Invalid Case Equivalence Classes	Boundaries and Special Cases	Notes
First number	-99 to 99	> 99 < -99 non-number expressions	99, 100 -99, -100 / : 0 null entry	
Second number	same as first	same as first	same	
Sum	-198 to 198			Are there other sources of data for this variable? Ways to feed it bad data?

Boundary Table as a Test Plan Component

- Makes the reasoning obvious.
- Makes the relationships between test cases fairly obvious.
- Expected results are pretty obvious.
- Several tests on one page.
- Can delegate it and have tester check off what was done. Provides some limited opportunity for tracking.
- Not much room for status.

- **Question, now that we have the table, do we have to do all the tests? What about doing them all each time (each cycle of testing)?**

Building the Table (in practice)

- Relatively few programs will come to you with all fields fully specified. Therefore, you should expect to learn what variables exist and their definitions over time.
- To build an equivalence class analysis over time, put the information into a spreadsheet. Start by listing variables. Add information about them as you obtain it.
- The table should eventually contain all variables. This means, all input variables, all output variables, and any intermediate variables that you can observe.
- In practice, most tables that I've seen are incomplete. The best ones that I've seen list all the variables and add detail for critical variables.

5. *Exploratory Testing*

- Exploratory testing involves simultaneously learning, planning, running tests, and reporting/troubleshooting results.
- A common goal of exploration is to *probe* for weak areas of the program.
- Proportions do (and should) vary across the industry. Here is my preference per week:
 - 25% of the group's time developing new tests
 - 50% executing old tests (including bug regression)
 - 25% on exploratory testing
- I recommend running new tests even in the very last days before releasing the product.
- Different testers do more or less well at exploratory testing. Don't assign 25/50/25 to everyone. Some people should do almost no exploration. Others should do it nearly full-time.

Second Cycle of Testing

- When you get a new build, look over the responses to the problems you've already reported:
 - **Immediately retest bugs that the programmers say they fixed.**

If you catch a bad fix right away, the programmer is likely to remember what she did. She can deal with it immediately. If you wait a few days, she'll forget details and wait until she has time to rethink and reinvestigate the problem..

- **Plan your responses to deferred and rejected bugs.**

Don't challenge every deferral. Pick your battles or no one will listen to you. Plan your approach—***if you're going to fight, win.*** Figure out what additional information would convince the other members of the project team that the problem should be fixed. In any case, think about other tests that the deferred bug suggests. The bug you reported might not be the worst of its line.

» refer to Testing Computer Software, pages 11-15

A Thought Experiment

- (I've never tried anything like this on Quicken. This example is purely hypothetical.)
 - Imagine testing Quicken. I understand that it can handle an arbitrarily large number of checks, limited only by the size of the hard disk. So, being a tester, you go buy the biggest whopping hard disk on the market (about 70 gig at the moment, I think) and then write a little program to generate a Quicken-like data file with many checks.
 - Now, open Quicken, load that file, and add 1 more check. Let's pretend that it works — sort of. The clock spins and spins while Q sorts the new check into place. At the end of four days, you regain control.
 - So you write a bug report— “4 days to add a check is a bit too long.”
 - The programmers respond— “Not a bug. Only commercial banks manage a billion checks. If you're Citibank, use a different program.”
- **Suppose that you decided to follow this up because you thought the programmers are dismissing a real problem without analysis. What would you do?**