

Black Box Software Testing

(Academic Course - Fall 2001)

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

Florida Institute of Technology

Section: 17 :

Exploratory Testing

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.

Black Box Software Testing

Exploratory Testing

- Several of these slides are from James Bach, with permission, or from materials co-authored with James Bach.
- Many of the ideas in these notes were reviewed and extended by my colleagues at the 7th Los Altos Workshop on Software Testing. I appreciate the assistance of the other LAWST 7 attendees: Brian Lawrence, III, Jack Falk, Drew Pritsker, Jim Bampos, Bob Johnson, Doug Hoffman, Chris Agruss, Dave Gelperin, Melora Svoboda, Jeff Payne, James Tierney, Hung Nguyen, Harry Robinson, Elisabeth Hendrickson, Noel Nyman, Bret Pettichord, & Rodney Wilson.

Exploratory Testing

- **Simultaneously:**
 - Learn about the product
 - Learn about the market
 - Learn about the ways the product could fail
 - Learn about the weaknesses of the product
 - Learn about how to test the product
 - Test the product
 - Report the problems
 - Advocate for repairs
 - ***Develop new tests based on what you have learned so far.***

Exploratory Testing

- **Tag line**
 - “Simultaneous learning, planning, and testing.”
- **Fundamental question or goal**
 - Software comes to tester under-documented and/or late. Tester must simultaneously learn about the product and about the test cases / strategies that will reveal the product and its defects.
- **Paradigmatic case(s)**
 - Skilled exploratory testing of the full product
 - Rapid testing
 - Emergency testing (including thrown-over-the-wall test-it-today testing.)
 - Third party components.
 - Troubleshooting / follow-up testing of defects.

Exploratory Testing

- **Strengths**

- Customer-focused, risk-focused
- Takes advantage of each tester's strengths
- Responsive to changing circumstances
- Well managed, it avoids duplicative analysis and testing
- High bug find rates

- **Blind spots**

- The less we know, the more we risk missing.
- Limited by each tester's weaknesses (can mitigate this with careful management)
- This is skilled work, juniors aren't very good at it.

Regression vs Exploration

- Regression
 - Inputs:
 - old test cases and analyses leading to new test cases
 - Outputs:
 - archival test cases, preferably well documented, and bug reports
 - Better for:
 - reuse across multi-version products
- Exploration
 - Inputs:
 - models or other analyses that yield new tests
 - Outputs
 - scribbles and bug reports
 - Better for:
 - Find new bugs, scout new areas, risks, or ideas

Doing Exploratory Testing

- Keep your mission clearly in mind.
- Distinguish between testing and observation.
- While testing, be aware of the limits of your ability to detect problems.
- Keep notes that help you report what you did, why you did it, and support your assessment of product quality.
- Keep track of questions and issues raised in your exploration.

Problems to be Wary of...

- Habituation may cause you to miss problems.
- Lack of information may impair exploration.
- Expensive or difficult product setup may increase the cost of exploring.
- Exploratory feedback loop may be too slow.
- Old problems may pop up again and again.
- High MTBF may not be achievable without well defined test cases and procedures, in addition to exploratory approach.

Models and Exploration

- We usually think of modeling in terms of preparation for formal testing, but there is no conflict between modeling and exploration. Both types of tests start from models. The difference is that in exploratory testing, our emphasis is on execution (try it *now*) and learning from the results of execution rather than on documentation and preparation for later execution.

Bubble (Reverse State) Diagrams

- **To troubleshoot a bug, a programmer will often work the code backwards, starting with the failure state and reading for the states that could have led to it (and the states that could have led to those).**
- **The tester imagines a failure instead, and asks how to produce it.**
 - Imagine the program being in a failure state. Draw a bubble.
 - What would have to have happened to get the program here? Draw a bubble for each immediate precursor and connect the bubbles to the target state.
 - For each precursor bubble, what would have happened to get the program there? Draw more bubbles.
 - More bubbles, etc.
 - Now trace through the paths and see what you can do to force the program down one of them.

Bubble (Reverse State) Diagrams

- **Example:**

- *How could we produce a paper jam (as a result of defective firmware, rather than as a result of jamming the paper?) The laser printer feeds a page of paper at a steady pace. Suppose that after feeding, the system reads a sensor to see if there is anything left in the paper path. A failure would result if something was wrong with the hardware or software controlling or interpreting the paper feeding (rollers, choice of paper origin, paper tray), paper size, clock, or sensor.*

Exploring Data Relationships

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

Data Relationship Chart

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

Procedural Relationships

- Pick a task
- Step by step, describe how it is done and how it is handled in the system (to as much detail as you know)
- Now look for ways to interfere with it, look for data values that will push it toward other paths, look for other tasks that will compete with this one, etc.

State Model-Based Testing

Notes from Harry Robinson & James Tierney

- By modeling specifications, drawing finite state diagrams of what we thought was important about the specs, or just looking at the application or the API, we can find orders of magnitude more bugs than traditional tests.
- Example, they spent 5 hours looking at the API list, found 3-4 bugs, then spent 2 days making a model and found 272 bugs. The point is that you can make a model that is too big to carry in your head. Modeling shows inconsistencies and illogicalities.
- Look at
 - all the possible inputs the software can receive, then
 - all the operational modes, (something in the software that makes it work differently if you apply the same input)
 - all the actions that the software can take.
 - Do the cross product of those to create state diagrams so that you can see and look at the whole model.
 - Use to do this with dozens and hundreds of states, Harry has a technique to do thousands of states.

» www.geocities.com/model_based_testing

Failure Model: Whittaker:

“The fundamental cause of software errors”

- Constraint violations
 - input constraints
 - such as buffer overflows
 - output constraints
 - computation
 - look for divide by zeros and rounding errors.
Figure out inputs that you give the system that will make it not recognize the wrong outputs.
 - storage violations
 - Really good for finding security holes

Interference Testing

- **We're often looking at asynchronous events here. One task is underway, and we do something to interfere with it.**
- **In many cases, the critical event is extremely time sensitive. For example:**
 - An event reaches a process just as, just before, or just after it is timing out or just as (before / during / after) another process that communicates with it will time out listening to this process for a response. (“Just as?”—if special code is executed in order to accomplish the handling of the timeout, “just as” means during execution of that code)
 - An event reaches a process just as, just before, or just after it is servicing some other event.
 - An event reaches a process just as, just before, or just after a resource needed to accomplish servicing the event becomes available or unavailable.

Interference: Interrupt

- **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

Interference: Change

- **Change something that this task depends on**
 - swap out a floppy
 - change the contents of a file that this program is reading
 - change the printer that the program will print to (without signaling a new driver)
 - change the video resolution

Interference: Stop

- 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: 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)
- Put the printer on local
- Put a database under use by a competing program, lock a record so that it can't be accessed — yet.

Interference: Compete

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

Error Handling

- **The usual suspects:**
 - Walk through the error list.
 - Press the wrong keys at the error dialog.
 - Make the error several times in a row (do the equivalent kind of probing to defect follow-up testing).
 - Device-related errors (like disk full, printer not ready, etc.)
 - Data-input errors (corrupt file, missing data, wrong data)
 - Stress / volume (huge files, too many files, tasks, devices, fields, records, etc.)

Troubleshooting

- **We often do exploratory tests when we troubleshoot bugs:**
 - **Bug analysis:**
 - simplify the bug by deleting or simplifying steps
 - simplify the bug by simplifying the configuration (or the tools in the background)
 - clarify the bug by running variations to see what the problem is
 - clarify the bug by identifying the version that it entered the product
 - strengthen the bug with follow-up tests (using repetition, related tests, related data, etc.) to see if the bug left a side effect
 - strengthen the bug with tests under a harsher configuration
 - **Bug regression: vary the steps in the bug report when checking if the bug was fixed**