# Black-Box Testing

Steven Zeil

September 15, 2013

## Contents

# 1 Testing

In this lesson, we explore how to do a thorough job of testing your programs.

If you've ever submitted a programming assignment that you thought was Ok and then been surprised to receive a poor grade for it, then it's a pretty good bet that the instructor had done a more rigorous job of testing your code than you had.

In the professional world, testing and debugging typically take far more time than actually writing the code. They may amount to as much as half the total development time with the requirements analysis, design, and coding taking up the remainder.

As a professional, it will be a mark of your professionalism and a factor in your company's reputation to catch bugs before the software is released. As a student, it's a matter of self-interest to do a good job of testing your code if you hope to get good grades.

**Why do we test?**

- To detect flaws in the software
  - (but what if we don't find any?)
- To gain confidence as to whether or not the system is usable

. . . . . .

## 1.1 Defintitions

**Testing and Debugging**

- *Testing* is the act of executing a program with selected data to uncover bugs.
- As opposed to *debugging*, which is the process of finding the faulty code responsible for failed tests.

. . . . . .

In practical terms, we don't start debugging until we know that something is wrong. How do we find out that something is wrong? Hopefully, we failed one of our own tests. If that's not how we found out, it's probably because someone using our software reported a bug. That, of course, makes us look bad.

And if someone else reported the bug, then before we can start debugging we have to figure out how to reproduce the bug, which means we are right back to testing, trying to find test data that reveal the bug so that we can figure out what's causing it.

**Failures, Faults, and Errors**

- A *failure* is an execution of a program on which incorrect output was obtained.
- A *fault* is a defect in the code that could lead to failures.
- An *error* refers to either
  - A defect in the output produced by a program
  - A mistake made by the software developer that resulted in a fault.

. . . . . .

**Test Cases and Suites**

- A *test case* is a description of a set of test inputs developed for a particular objective.
  - Could be so detailed as to give the actual input data, but often simply describes the desired input data.
- A *test suite* is a collection of test cases.

. . . . . .

## 1.2 Choosing Test Data

**Strategies for Choosing Tests**

Testing traditionally can be conducted in three styles

- *Black-Box* testing
  - Try to choose "smart" tests based on the requirements, without looking at the code.
- *White-Box* testing
  - Try to choose "smart" tests based on the structure of the code, with minimal reference to the requirements.
- *Random* testing
  - Try to use directed random selection to choose tests that are "representative" of how the program will be used.



. . . . . .

We'll discuss the black-box testing of these today and white-box later this semester.

Random testing is not used as often. It sometimes is useful as a way check against unwarranted assumptions by the developers and testers ("Oh, I never thought of even trying *that*!"). But its primary use is in large projects where statistics collected during random testing are plugged into models that are used to predict how reliable the software will be if released in its current state or how much longer we will need to test before we reach a desired level of reliability.

# 2 Black Box Testing

**Black-Box Techniques**

There are a number of well-known BB strategies.

- Representative Inputs
- Functional Coverage
- Boundary-Values Testing
- Special-Values Testing

A good test BB suite will include all of these.

In fact, these should become second nature to most programmers.

. . . . . .

## 2.1 Representative Inputs

**Representative Inputs**

Choose at least one test that is a "typical" input to the system when in real use.

- This may require some study of how people will use the system.
  - If you are replacing an existing system, or automating a task that people have been doing manually, you may be able to collect "real" inputs.

. . . . . .

**Example: the Auction Program**

What would we use as "typical" or "representative" inputs to our auction program?

. . . . . .

Well, the auction program description contains some sample input, and that's certainly worth using, but it's really not clear how representative that's intended to be.

But there's also mention of existing auction systems, so we might try to collect some sample sessions from some of those.



**Test Specifications**

Software development projects manage testing via a couple of standard documents:

- A *test plan* is a document describing what will be tested, who will do it, and the procedures they will follow.
- A *test case specification* or *test specification* describes the test cases for an item to be tested.
- These are often confused with one another.
  - Test plans are for management.
  - Test specs are for the software developers.

. . . . . .

We'll demonstrate the development of a test specification document, following (roughly) the format of the IEEE 829 Standard for Software and System Test Documentation.

**Starting the Auction Program Test Spec**

**1. Module Overview**
The auction program resolves bids received at an online auction site.
**1.1 Inputs** Three files, one describing the items up for bid on a single day, another describing the registered bidders, a third describing bids received.
**1.2 Outputs** List of auction winners, written to standard output.
**2. Test Data**
**Representative Input**

1. At least one typical input as described in the requirements document.

That last item is our first *test case*.

. . . . . .

## 2.2 Functional Coverage

This is a bit of a misnomer. the "function" in "Functional Coverage" has nothing to do with the idea of a C++ function. Instead it really pertains more to "functionality".

If you look though most program requirements, you'll see that the program is probably supposed to carry out a variety of different behaviors under various circumstances. Sometimes these behaviors are described in terms of input conditions:

> For any input $x$, if $x \geq 0$, return $x$ but if $x < 0$, return $-x$.

Here we have clearly drawn a distinction between different behaviors depending upon the input values.

In other cases, distinct behaviors are most easily identified by looking at the output requirements:

> This program reads a file name from the input. It should print a single line of output
>
> *filename file-size-in-kilobytes*
>
> if the file exists but should print
>
> *filename* does not exist.
>
> if the file cannot be found.

Here we can tell from the distinct output specifications that we have two separate behaviors going on.

Now, from a purely theoretical point of view, distinct behaviors must have both distinct input pattern and distinct output patterns. I could just as easily argue that, in the above example, the two behaviors are associated with the input cases of "the input named an existing file" and "the input named a non-existent file". But sometimes the people writing the requirements will find it easier to focus on the input conditions and sometimes on the output cases. As a tester (and coder), you have to roll with whatever they give you.

Sometimes, a behavior may be described in purely internal terms:

> Apply the formula
>
> ```
> x = x  - (x*x - N) / 2
> ```
>
> up to 1000 times until $x$ changes by no more than $\pm 0.001$. If, after 1000 iterations, $x$ is still changing by more than that, abort the program with message
>
> ```
> Calculation does not converge.
> ```

In this case, if $x$ is not an input but some value computer internally, it might be quite difficult to figure out what inputs would lead to this "does not converge" behavior. Nonetheless, as a tester, you will want to try to do so.

**Functional Coverage**

Choose at least one test that covers each distinct "behavior" described in the requirements.

- Different functions performed by the program
- Different types or ranges of input that get treated or described separately.



- Different types or ranges of output that get treated or described separately.

. . . . . .

Don't make the common mistake of that different parts of a program's calculations or outputs are distinct behaviors. For example, given a requirement

> This program reads a file name from the input.
> It should print that file name.
> If the file exists, it should then print the size of the file (in kilobytes), on the same line.
> If the file does not exist, it should print " does not exist." on the same line as the file name.

We only have two distinct behaviors here, not three. The printing of the file name is not a distinct behavior from the behavior of printing the file size or the "does not exist" message. It's simply a component of the behavior in each of the other two cases. How do we know? Because there is no distinct set of inputs on which file names are (and are not) printed.

**Example: Auction Functional Coverage**

**Question:** What are the distinct behaviors associated with the auction program?

- Look particularly at the discussion of how bids are handled and
- at the possible outputs for each auction

. . . . . .



- Bids can be
  - on time or late
  - above reserve or not
  - more than a bidder can afford or not
- Output can be
  - Announcement of a winner
  - no winner for an item

**Continuing the Auction Program Test Specification**

**2. Test Data**
**Representative Input**

1. At least one typical input as described in the requirements document.

**Functional Coverage**

2. Highest bid for some item qualifies to win.
3. Highest bid for some item is too late but otherwise qualifies to win.
4. Highest bid for some item is below the reserve price but otherwise qualifies to win.
5. Highest bid for some item is above the bidder's account balance but otherwise qualifies to win.
6. No qualifying bids received for some item

. . . . . .

Note: many test cases can be satisfied by a single set of test input run. That's OK, as long as they don't intuitively interfere.

For example, by packing lots of auctions into one set of input files, we might well be able to cover all of the above cases with one test run.

Now whether or not that's a good idea is another matter. If you carry it to an extreme, you wind up with tests that are hard to check. And if something goes wrong and you have to start debugging, you won't be happy if you have to wade through 99 correctly handled behaviors to get the buggy number 100.

## 2.3 Boundary Values Testing

**Boundary Values Testing**

A.k.a., *Extremal Values Testing*

Choose as test data the largest and the smallest values for each input and for each "functional behavior" range

. . . . . .

**Seeking Boundaries**

- A common mistake is to simply look at the input data and to try to choose the largest and smallest inputs possible.
  - Nothing wrong with using that data
  - But it shortchanges the test set quite a bit.
- Instead, seek the boundaries of the *functional test cases* that you specified earlier.

. . . . . .

**Sample Boundary Tests**

Suppose that we are testing an implementation of the absolute value function `abs(x)`

- Distinct behaviors for $x \geq 0$ and $x < 0$
  - So, during functional coverage, we might use tests $x = 12$ and $x = -1015$
- But boundary value testing would suggest that we look at
  - An extremely large $x$ and $x = 0$: the boundaries of the $x \geq 0$ behavior
  - An extremely large negative number and $x = -1$: the boundaries of the $x < 0$ behavior

. . . . . .

The test cases 0 and $-1$ are useful because they give us a chance of detecting common errors like using the wrong relational operator.

Notice that, if we have done a thorough job of listing the functional cases for a program, then we will get the “largest and smallest possible inputs” cases automatically when we list out the largest and smallest cases for each distinct behavior.



**Example: Auction Extremal Values**

We had functional cases

- Highest bid for some item qualifies to win.
- Highest bid for some item is too late but otherwise qualifies to win.

Concentrating on the time input, we would focus on the boundary between being on time and too late:

- Bids can be
  - exactly on time
  - one second late

. . . . . .

**Example: Auction Extremal Values (cont.**

We also have some explicit limits on the legal inputs, e.g.:

- #items is non-negative
- auction end time: hours is 00..23, minutes is 00..59, seconds is 00..59

This leads to some additional boundary cases:

- #items of zero
- auction ending at midnight (00:00:00)
- auction ending one second before midnight (23:59:59)

. . . . . .

**Continuing the Auction Program Test Specification**

**2. Test Data**

**Representative Input**

1. At least one typical input as described in the requirements document.

**Functional Coverage**

2. Highest bid for some item qualifies to win.
3. Highest bid for some item is too late but otherwise qualifies to win.



4. Highest bid for some item is below the reserve price but otherwise qualifies to win.
5. Highest bid for some item is above the bidder's account balance but otherwise qualifies to win.
6. No qualifying bids received for some item

**Boundary Values**

7. Winning bid arrives at same second as auction ends
8. Winning bid arrives one second after auction ends
9. Winning Bid exactly matches reserve price
10. Winning Bid one cent below reserve price
11. Winning Bid exactly matches bidder's account balance
12. Winning Bid one cent above bidder's account balance
13. Number of items is zero
14. Number of items is large
15. Reserve price is zero
16. Reserve price is large
17. Auction end time 00:00:00
18. Auction end time 23:59:59
19. # bidders is zero
20. # bidders is large
21. bidder with account balance zero
22. bidder with large account balance
23. # bids is zero
24. # bids is large
25. amount of bid is zero
26. amount of bid is large



27. bid at time 00:00:00
28. bid at time 23:59:59

. . . . . .

## 2.4 Special Values Testing

**Special Values Testing**

Choose as test data those certain data values that just tend to cause trouble. Programmers eventually develop a sense for these. They include

- For integers: -1, 0, 1
- For floating point numbers: -e, 0, +e, where "e" is a very small number
- For strings: the empty string, strings containing only blanks, strings containing no alphabetic characters

. . . . . .

**What is Special?**

- As we move to other data structures, we may develop a suspicion of other special values, e.g.,
  - For times of the day: midnight, noon
  - For containers of data: an empty container

Special values and Boundary values often overlap. That's OK.

. . . . . .

One of my favorite stories about bugs that (should have) been caught by boundary and special values testing comes from early testing of the software controllers for the F-16 jet fighter. During fight simulations (thankfully!), one pilot decided to fly the simulated plane across the equator. A sign error in handling southern latitudes caused the control software to decide that the plane was upside down, and to attempt to roll the plane over to compensate.

**Example: Auction Special Cases**

⋮

**Special Values**

29. *Number of items is zero*
30. Number of items is one
31. Item name has no alphabetic characters
32. *Item reserve price is zero*
33. *Auction ends at midnight*
34. Auction ends at noon
35. *Number of bidders is zero*
36. Number of bidders is one
37. *Bidder's balance is zero*
38. Bidder's name has no alphabetic characters
39. *Number of bids is zero*
40. Number of bids is one
41. *Amount bid is zero*
42. *Time of bid is* midnight
43. Time of bid is noon

. . . . . .

**Completed Test Specification**

After eliminating *duplicates* from the preceding list,

**Test Specification** Auction Program
**1. Module Overview**
The auction program resolves bids received at an online auction site.
**1.1 Inputs** Three files, one describing the items up for bid on a single day, another describing the registered bidders, a third describing bids received.
**1.2 Outputs** List of auction winners, written to standard output.
**2. Test Data**
**Representative Input**

1. At least one typical input as described in the requirements document.

**Functional Coverage**

2. Highest bid for some item qualifies to win.
3. Highest bid for some item is too late but otherwise qualifies to win.
4. Highest bid for some item is below the reserve price but otherwise qualifies to win.
5. Highest bid for some item is above the bidder's account balance but otherwise qualifies to win.
6. No qualifying bids received for some item

**Boundary Values**

7. Winning bid arrives at same second as auction ends
8. Winning bid arrives one second after auction ends
9. Winning Bid exactly matches reserve price
10. Winning Bid one cent below reserve price
11. Winning Bid exactly matches bidder's account balance
12. Winning Bid one cent above bidder's account balance
13. Number of items is zero
14. Number of items is large
15. Reserve price is zero
16. Reserve price is large
17. Auction end time 00:00:00
18. Auction end time 23:59:59
19. # bidders is zero
20. # bidders is large
21. bidder with account balance zero
22. bidder with large account balance

23. # bids is zero
24. # bids is large
25. amount of bid is zero
26. amount of bid is large
27. bid at time 00:00:00
28. bid at time 23:59:59

**Special Values**

29. Number of items is one
30. Item name has no alphabetic characters
31. Auction ends at noon
32. Number of bidders is one
33. Bidder's name has no alphabetic characters
34. Number of bids is one
35. Time of bid is noon

. . . . . .

## 3 Illegal Inputs

**Illegal Inputs**

Earlier, we observed that the input requirement:

#items is non-negative

led to a boundary value test case:

- #items of zero

I did *not* add a test case

- #items is −1

Why not?

. . . . . .



**Illegal Inputs**

If we have an input requirement

> #items is non-negative

then the input `#items == -1` is *illegal*.

- You *cannot* test a program on illegal inputs.

  Notice that I did not say "should not". I said "cannot". It can't be done.

  Therefore...

- Trying to test a program on illegal inputs is just a waste of time and effort.

. . . . . .

You may have been told something differently by other teachers or read something different elsewhere.

I don't care.

They are either wrong, or you misunderstood what they were talking about.

There are people who make statements like "a good program never crashes" or "a good program should always behave sensibly on any input". Setting aside the fact that the first is impossible and the second relies on a definition of "sensible" that no two people will ever agree upon, these kinds of statements are really aimed at how to write a good requirements statement for programs. They have nothing at all to do with how we code programs or how we test them.

**Why CAN'T we test illegal inputs?**

- A test is useless if you can't tell whether the program passes or fail is.
- By definition, a program can do *anything* on an illegal input.
  - It could print an error message and abort
  - It could print an error message and try to keep running
  - It could pop up a menu asking the operator what it should do.
  - It could send email to all of your friends complaining about the lousy input that you tried to force it to process.
  - It could contact the nearest SAC base and call in an airstrike on your keyboard.

On an illegal test input, no matter the program actually does, it passes the test *by definition*.

. . . . . .



**Illegal versus Unusual Inputs**

Don't confuse illegal inputs with unusual inputs

- If I ask you to write a function `mySqrt(double x)` that

> computes the square root of any non-negative $x$

then negative inputs are *illegal*.

  - The function has no specified behavior in that case. Any program that calls this function must make sure it never supplies negative input values.

- If I ask you to write a function `mySqrt(double x)` that

> computes the square root of any non-negative $x$ but returns -1.0 when given a negative $x$

then negative inputs are not illegal.

  - We might expect them to be rare, but the behavior of the function is fully specified in that case,
  - so it's perfectly OK for programs to call the function with negative numbers.

. . . . . .

**How do you test a program on. . .**

- . . . an illegal input?
  - You can't. Whatever the program does is correct behavior for that program.
- . . . an unusual input?
  - Check to be sure that it does whatever the requirements document says.
  - Such tests should emerge naturally from functional coverage
    - Each "kind" of unusual input is a distinct behavior.

. . . . . .



# 4 Some General Guidelines for Testing

**Choosing the Test Data**

Within the limits of the test case specification,

- Choose the simplest inputs you can
  - Easy to come up with the input
  - Easy to show it satisfies the test case
  - Easy to figure out what the correct output is supposed to be
  - Easy to see if the output actually is correct

. . . . . .

**Choosing the Test Data (cont.)**

- OK to overlap test cases on the same input set
  - as long as they don't interfere with or "hide" one another
  - but usually better to have lots of simple tests than a few large ones
- KISS!
  - Because we all get a little bit "stupid" after staring at the screen for hours

. . . . . .

**Conducting Tests**

- Get out of the habit of typing all your tests by hand
  - Discourages people from running more than a few tests
  - Makes it hard to repeat tests
- Alternatives:
  - redirection (CS252, but also possible in Windows)
  - copy-and-paste from text file into running program
  - redesign program to allow file names in command line

Many of these are covered in the earlier Labs.

. . . . . .