Debugging – Using Hypotheses to Track Down the Culprit

Steven Zeil

July 18, 2013

Contents

1 Rules of Debugging

Recap – Rules of Debugging

1. You can't debug what you don't understand
2. Make it fail...
   (a) Make it fail *reliably*
   (b) Make it fail *easily*
   (c) Make it fail *visibly*
3. Track down the culprit
   (a) Don't Guess, Hypothesize
   (b) Divide and Conquer
   (c) If you didn't fix it, it ain't fixed!

In this lesson we focus on tracking down the culprit.

. . . . . .

An actual message received from a student

Professor Zeil,

Upon running my code, I found several errors that I have been unsuccessful in solving. The most glaring error is that when I try to use the "Tile" option under the view menu, I get an error concerning a division by zero. Looking back at where the error references the code, it seems to be happening at the point where there is division by the bounding box height and width. This leads me to believe that my bounding box function for the Composite is flawed. This is not unlikely considering my inexperience with Java, but by my estimation it should have worked. I used the union function to combine the bounding boxes of different shapes, so there should be no reason that the bounding box returned by the Composite would be zero. Unless there are no shapes in the list to begin with, but that should not be the case either. When I use the "Draw Composite" option under the view menu, shapes are drawn, so there should be shapes in the Composite private member field (I am using a vector to store the shapes). Additionally,

when I select the "Move Composite" or "Reflect Composite" options, nothing is displayed, but there is no error message given either. Any assistance you can provide would be greatly appreciated.

. . . . . .

This email was sent in regards to an assignment involving manipulating pictures composed of many component shapes.

A key idea in such applications is that of a *bounding box*, the smallest rectangle that can enclose a set of shapes. Computing the bounding box was one of the first steps required for many of the picture manipulations.)

What's Wrong with That?

- The message shows a lot of good thought about what might be going wrong.
  - And a lot more detail than many students supply when asking for help
- But because there is no follow-up, it's all *guesswork*.

. . . . . .

Wasted Effort

Some of the thought going into those guesses may have been wasted effort:

> "There is nothing as deceptive as an obvious fact" – Sherlock Holmes, *The Boscombe Valley Mystery*

This leads me to believe that my bounding box function for the Composite is flawed...the bounding box returned by the Composite would be zero.

- This whole line of reasoning takes for granted that the problem is a zero bounding box.
- And that the problem occurred in the function where the box was first computed.

That's an awful lot of effort going into something that might not even be remotely true.

. . . . . .

Continuing in that vein

I used the union function to combine the bounding boxes of different shapes, so there should be no reason that the bounding box returned by the Composite would be zero. Unless there are no shapes in the list to begin with, but that should not be the case either.

"It is a capital mistake to theorize before one has data." – Sherlock Holmes, *A Study in Scarlet*

- Since we don't know yet *if* the bounding box is zero, why speculate on how it got that way?

This leads us to the key rule of this lesson...

. . . . . .

2 Don't Guess, Hypothesize

Don't Guess, Hypothesize

The problem with the above-quoted student is that he was making *guesses* instead of *hypotheses*.

- An "*hypothesis*" is a guess that can be tested to see if it is true or not.
- In most cases by instrumenting the code to print the critical information at the appropriate location.

. . . . . .

Guessing

Looking back at where the error references the code, it seems to be happening at the point where there is division by the bounding box height and width.

- Assumes (guesses) that height and width are interchangeable in this bug

. . . . . .

More Guessing

This leads me to believe that my bounding box function for the Composite is flawed.

- Guesses that the problem lies in the function that computes the bounding box
- Does not consider alternatives
  - Could something be altering the box later?
  - Could there be pointer errors overwriting the memory occupied by the bounding box variable?

. . . . . .

Still More Guessing

by my estimation it should have worked. I used the union function to combine the bounding boxes of different shapes, so there should be no reason that the bounding box returned by the Composite would be zero.

- Guesses that `union` function is working and is being used appropriately

Unless there are no shapes in the list to begin with, but that should not be the case either.

- Guesses that list is not empty.

. . . . . .

Won't the Guesswork Ever Stop?

When I use the "Draw Composite" option under the view menu, shapes are drawn, so there should be shapes in the Composite private member field

- Guesses that ADT data member is not empty during "draw" operations

. . . . . .

Can you phrase that in the form of a question?

Additionally, when I select the "Move Composite" or "Reflect Composite" options, nothing is displayed, but there is no error message given either.

- Guesses that this symptom is related to he problem under discussion
- As opposed to the above-mentioned ADT data member simply being empty during "move" and "reflect" operations

. . . . . .

2.1 Hypothesizing

Hypothesizing

The frustrating thing about all that guesswork is that it would have been easy to check and see which of those guesses were actually true:

Looking back at where the error references the code, it seems to be happening at the point where there is division by the bounding box height and width.

- So add debugging output to print the height and width.

. . . . . .

Guess + Test = Hypothesis

This leads me to believe that my bounding box function for the Composite is flawed.

- So print the return values from the bounding box function and see if the output really is flawed or not.

. . . . . .

Why Engage in Idle Speculation?

by my estimation it should have worked. I used the union function to combine the bounding boxes of different shapes, so there should be no reason that the bounding box returned by the Composite would be zero.

- Maybe it's **not** zero!
- So print the bounding box and see if it is zero or not.
  - If it is, add debugging output after each call to the `union` function to see when it becomes zero.

. . . . . .

Hypothesize

Unless there are no shapes in the list to begin with, but that should not be the case either.

- So print the list, or at least the length of the list.

. . . . . .

Hypothesize, Hypothesize

When I use the "Draw Composite" option under the view menu, shapes are drawn, so there should be shapes in the Composite private member field

- So print out that field and see what's in there.

. . . . . .

Hypothesize, Hypothesize, Hypothesize

Additionally, when I select the "Move Composite" or "Reflect Composite" options, nothing is displayed, but there is no error message given either.

- And print the same field in these functions as well.

. . . . . .

2.2 Debugging as a Series of Science Experiments

Debugging as a Series of Science Experiments

As you trace backwards along the path from failure to possible faults,

- Think of each step as a mini-science experiment.
  - Hypothesize that X is wrong.
- Then perform the experiment.
  - Examine the value of X, via debugging output or an automated debugger.

> "You see, but you do not observe. The distinction is clear." – Sherlock Holmes, *A Scandal in Bohemia*

. . . . . .

3 Narrowing the Search

Narrowing the Search

- In a large system with many components, you do not want to waste time studying all the parts that *are* working
- Narrow the search as quickly as possible
- We've already discussed the general flow of debugging:
  - choose the simplest test that reliably illustrates the failure
  - work backwards from known bad states

. . . . . .

Divide and Conquer

If you can find a location along a probable backwards infection path that

1. Is easily checked for infection

2. Divides the system/path into roughly equal chunks

then test the hypothesis that the failure occurs before that location.

- Examine the state at that location to see if it is infected.
- If it is, you don't have to examine the back half of your system/path.
- If it is not, you don't have to examine the front half of your system/path.

You can save a lot of time and effort by repeatedly cutting your search space in half.

Essentially, this is the "binary search" approach to debugging.

What if a Problem Has Multiple Possible Causes?

If you find that there are multiple potential causes for the observed failure, you may have to test hypotheses for each in turn.

> "When you have eliminated all which is impossible, then whatever remains, however improbable, must be the truth." – Sherlock Holmes, *The Blanched Soldier*

- If you find the cause, great!
- If not, you have still narrowed the possibilities

4 If you didn't fix it, it ain't fixed!

"If you didn't fix it, it ain't fixed!"

(Quoted from *Debugging* by David Agans)

If an error goes away, and you don't understand why, don't trust that it is really gone.

> ""Is there any point to which you would wish to draw my attention?" "To the curious incident of the dog in the night-time." " – Sherlock Holmes, *Silver Blaze*

Grounds for Suspicion

If an error goes away, and you don't understand why, don't trust that it is really gone.

> ""The dog did nothing in the night-time."
> "That was the curious incident," remarked Sherlock Holmes." – Sherlock Holmes, *Silver Blaze*

- Is the bug intermittent?
  - If you undo your latest "fix", does the bug return?
- You might merely have altered the input set on which it manifests.
  - E.g., changing critical "size" constants may defer overflow problems but not eliminate them.
- Bugs can hide one another (on some but not all inputs)

. . . . . .