Lab 3: Debugging

Introduction: Of Moths, Wolves, and Rubber Ducks

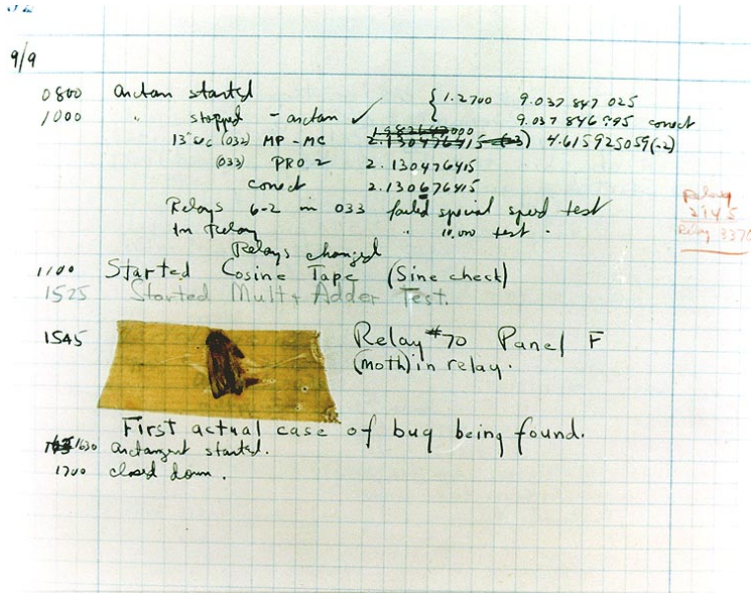

Figure 27. Grace Hopper's famous log entry.

Debugging is the term used in the software engineering industry to describe the process of locating and correcting faults in software. It is a term that comes from outside of software engineering, and its use goes back to at least Thomas Edison's days. One of the first documented uses of the word to describe the process in computer science goes back to Rear Admiral Grace Hopper's use to describe pulling a moth out of a computer circuit in 1947; she actually found a moth in the computer hardware. Her famous log entry is depicted in Figure 27.

Bugs can be divided into two broad categories. Syntax bugs, by far the easiest to catch, are mistakes in the "spelling" of code; compilers are generally very good at finding these types of bugs. Semantic bugs, conceptual errors in the code, are much more difficult to find. Fortunately there are tools and techniques that we can familiarize ourselves with to aid in debugging these types of bugs.

This lab is designed to introduce you to some debugging techniques, and introduce you to two debugging tools: a code debugger, and a web page debugging tool.

Lab 3 Preparatory Work

About Bugs

Research has been conducted around software bugs, their causes and types, and effective means of removing and preventing them. Research seems to indicate that knowing about the nature of bugs helps debugging.

Researchers at Yale identified seven common causes of bugs committed by beginning programmers[1]. Some of the errors identified include off-by-one errors (looping once too few or many times), plan-dependency problems (putting code in the wrong part of a complicated structure), and negation and whole part problems (forgetting to apply DeMorgan's theorem when needed).

Research has shown that, surprise, learning about bugs can help new programmers debug better.

Ducks and Wolves: Two Basic Strategies for Debugging

Figure 28. Your bug is a howling wolf.

There are two basic strategies that can help you debug your software. The first is a debugging algorithm developed by Edward Gauss, not surprisingly at the University of Alaska[2]. Applying this algorithm results in your bug becoming isolated, an

[1] McCauley, et. al., "Debugging: a review of the literature from an educational perspective." *Computer Science Education*. Vol. 18 No. 2, June 2008, 67-92

[2] Gauss, E., "The 'Wolf Fence' algorithm for debugging." Communications of the ACM, Vol. 25 No. 11, Nov. 1982, 780—

important step in any debugging process. He states it like this. Pretend your bug is a howling wolf and your code is the entire territory of Alaska. Then:

1. Let A be the territory known to contain the wolf (initially all of Alaska).
2. Construct a fence across A, along any convenient natural line that divides A into B and C.
3. Listen for the howls; determine if the wolf is in B or C.
4. Go back to Step 1 until the wolf is contained in a tight little cage.

But what if you find your bug and you're not sure why its there? The next technique helps you try to reveal your bug by explaining what it was that got you there in the first place.

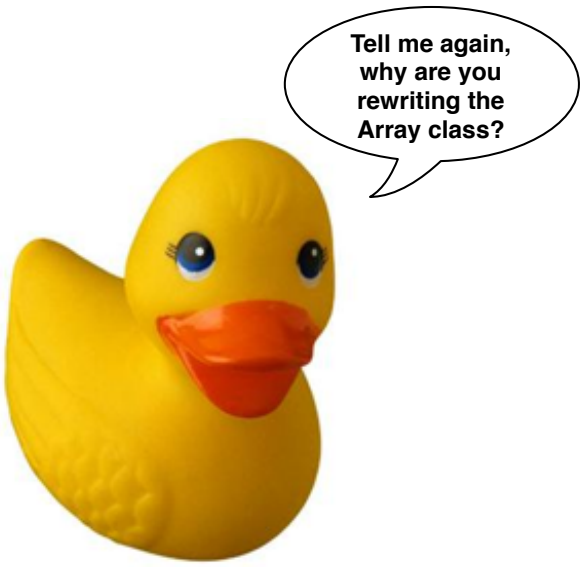

Figure 29. Ducks eat bugs.

A strategy called "Rubber Ducking" is based on the premise that you have a rubber duck with you at all times when programming. You should pause in your coding, and explain clearly (rubber ducks need things to be kept simple) what it is you are doing. The concept is that this helps you keep your goal (why was I writing this code?) in mind as you proceed with your coding; losing track of the coding goal is a common reason for bugs.

Lab 3 Classroom Work

Debugging Java with the NetBeans Debugger

Most beginning programmers learn the technique of printing out variable values from within their program to determine what the cause of a bug might be. While this is a tried-and-true method, it is time consuming. Over the years, debugging tools have developed to help ease the debugging process.

For this example, you will use the NetBeans debugger. This debugger has most of the tools that any other debugging programming might contain, and is a good introductory debugger to work with.

1. Open NetBeans. Create a new Java application project named DebugExample.

2. Browse to http://thecity.sfsu.edu/~csc412/DebugExample.java, and copy the code into the project you created. Make sure that the package and class names correctly match your project settings.

```
run:
  1  2  3  4  5  6  7  8  9 10 **
  2  3  4  5  6  7  8  9 10 **
  3  4  5  6  7  8  9 10 **
  4  5  6  7  8  9 10 **
  5  6  7  8  9 10 **
  6  7  8  9 10 **
  7  8  9 10 **
  8  9 10 **
  9 10 **
 10 **
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 30. Desired output from DebugExample program.

3. The desired output of the program is depicted in Figure 30. Run the program. The output you get should be different, and in fact the run should generate a runtime error. We will now attempt to find the problem using the NetBeans debugger.

4. If you examine the output that was generated during your run, you can see that the program is trying to work, there are at least some numbers printing and some asterisks.

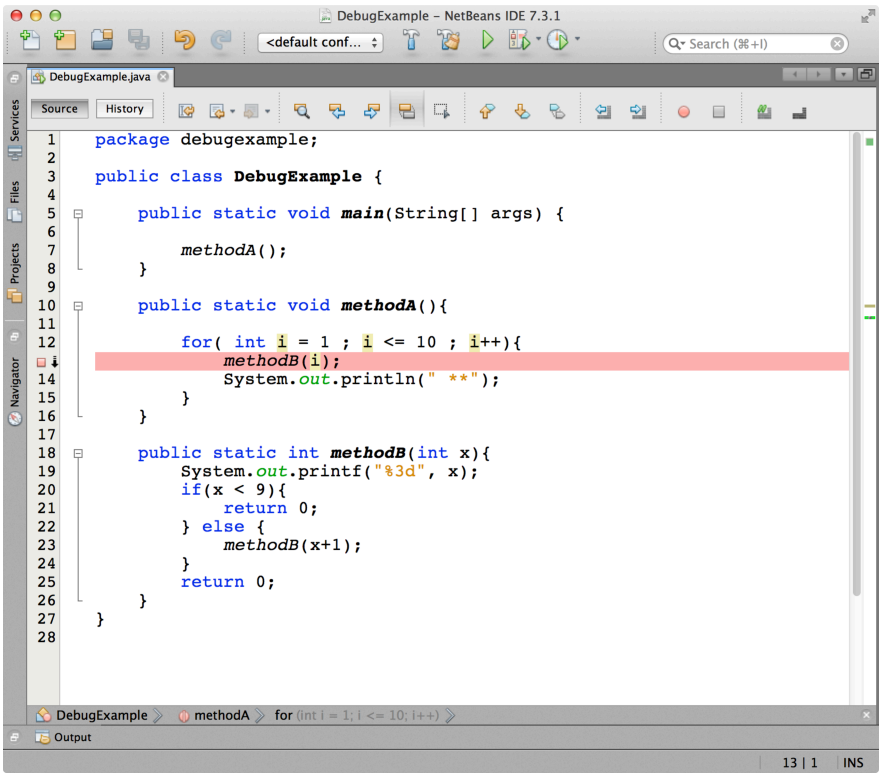

Figure 31. Breakpoint set on line 13.

5. Set a breakpoint at line 13 in the source code. When debugging a program, a breakpoint is where the program pauses and allows you to inspect what is going on in the program. To set a breakpoint, click on the 13 in the bar of numbers. It should then appear as depicted in Figure 31.

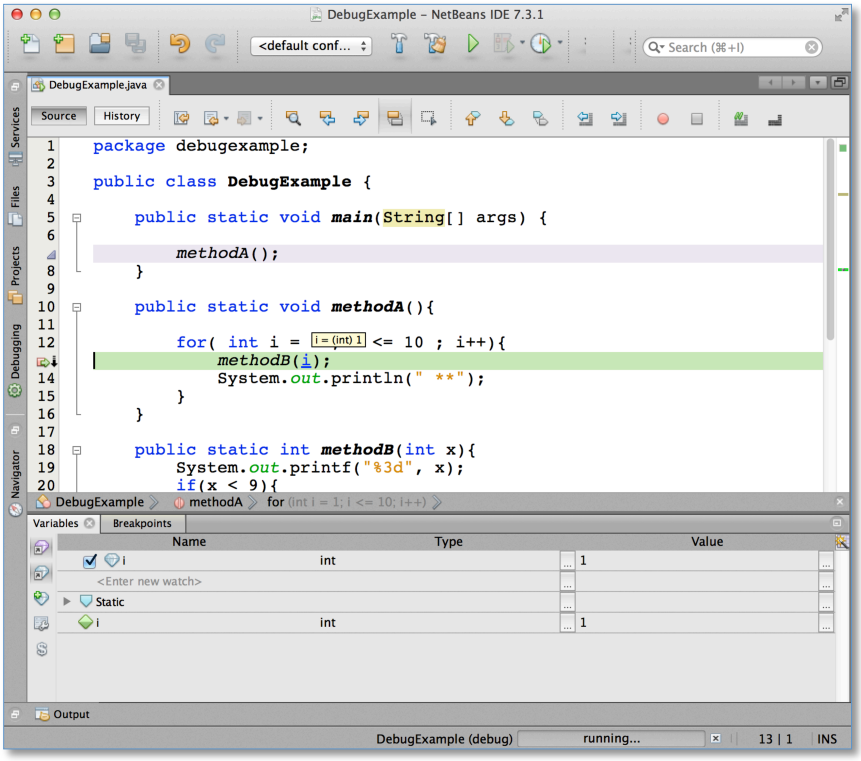

Figure 32. The value of i at the first break.

6. Run the program in debug mode. To do this, from the menu select Debug > Debug Main Project (or either ⌘F5 or Ctrl⌘F5 on a Mac, or ctrl-F5 in Windows). Your program should now be paused at line 13. If you hover your cursor over the variable i in line 13, you should see a value pop up, indicating the current value of i. You can also see i's current value in the variables window on the bottom (Figure 32).

7. Now, examine what happens the next time around in the loop. To see this you want to "continue" program execution. To do this, from the menu select Debug > Continue (or F5 on a Mac and in Windows). Notice how the value of i changes. Continue execution a number of times until the value of i reaches 5.

8. Examine the output window. You should see 4 rows of output text.

9. A feature among many debuggers is the ability to change the value of a variable while the program is running (in debug mode). In the variables pane, click on the button with an ellipsis (three dots) to the right of i, and enter the value 3.

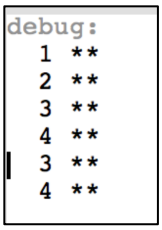

Figure 33. Output after changing the loop control variable.

10. Continue execution of the program through two more loops (until i is 5), and examine the output window's contents. It should appear as if the loop backed up two steps (Figure 33). This is, of course, exactly what happened since you modified the value of the loop control variable in step 9.

11. Another feature that most debuggers offer is the ability to allow the program to run one line of code at a time, pausing after each line. This is called stepping, of which there are different types:

    Step Over (Debug > Step Over or F8) – Executes the highlighted line. If the line contains a method call, it runs the method without descending into the method, pausing at each statement in the method.

    Step Into - (Debug > Step Into or F7) – Executes the highlighted line. If the line contains a method call, the debugger descends into the method, and pauses on the first line in the method.

    Step Out - (Debug > Step Out or ⌘F7 / ctrl–F7) – Executes the highlighted line. If the line is in a method, the debugger continues executing the remaining lines in the method, returns from the method, then pauses in the location that the method was called.

12. Keep stepping (10 or 20 times) through the program using step over and step into (remember to step over when you get to the println statement). Observe what is happening in the variables window, and observe what is getting executed each time you seem to "loop". Remember, stepping into means that you descend into the method. What type of method is methodB (hint: methodB calls itself)?

13. Using the debugger, set the value of x to 0 before you enter methodB again, keep stepping into the method, observing what happens. When you have an idea what is happening, you can terminate the program in the debugger (Debugger > Finish Debugger Session).

14. Can you figure out what may be wrong with the program? If you still can't figure it out using the debugger, scroll down to the answer in the references section at the end of the lab.

15. Correct the error, then run the program normally, and observe the output. It should appear as in Figure 30.

Debugging Web Pages using Chrome Developer Tools

During Lab 2, many people had difficulties with getting the CSS to cooperate. The tools demonstrated here can help you debug web pages much more easily than just adjusting values in your CSS and reloading the page (the web equivalent to printing and running your program code).

Most web browsers now provide tools to allow you to understand how the browser interprets and renders a web page and why. Google's Chrome browser provides Google's Dev Tools for this purpose. In this part of the lab you will modify your class page's CSS using Dev Tools, but this is just the beginning. Dev Tools prvides many powerful tools that significantly aid in web page development. You are encouraged to experiment with and do further reading about Dev Tools.

1. Open Chrome. If you don't have Chrome installed on your computer, go to http://google.com/chrome, and download and install it.

2. Go to your *about* webpage (e.g. http://thecity.sfsu.edu/~csc412/about.html)

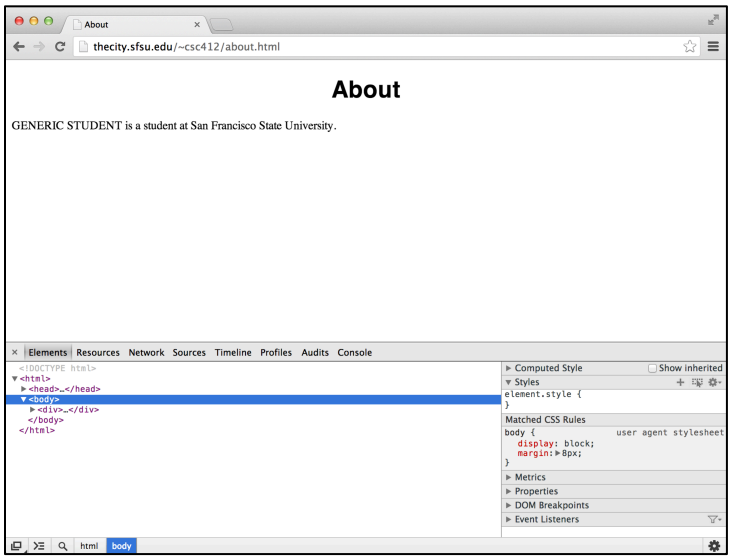

Figure 34. Chrome with Developer Tools enabled.

3. Open the developer tools (View > Developer > Developer Tools or `option⌘i` on Mac, Tools > Developer Tools or `ctrl-⇧-i` in Windows). Panes should appear at the bottom of the page and should appear as in Figure 34.

4. Move your cursor up and down (no clicks) over the elements in the lower-left pane. You should observe different sections of your web page being highlighted. What you are seeing is a correlation between what HTML element corresponds to what part of your page.

5. The right hand pane contains information about the current state of your page, including the current styles in use in your page. Look through the different information that is available to you.

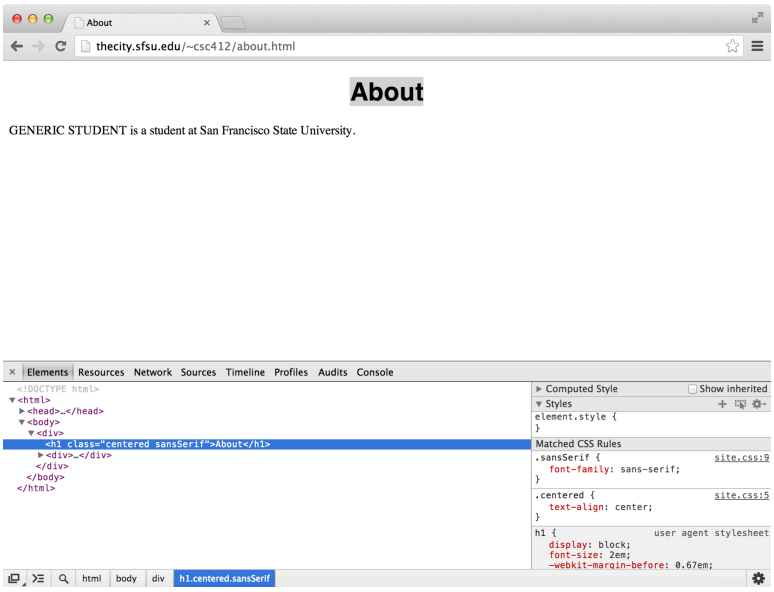

Figure 35. Inspecting an element of a webpage in Chrome.

6. In the upper pane, right click on the word About, and select Inspect Element. You should see the About title's html element selected on the lower-left pane, and the matching CSS rules in the lower-right pane (Figure 35).

7. If you mouse over the line containing font-family under Matched CSS Rules, a checkbox will appear. Click on the checkbox so the box is cleared, and notice what happens to your title. Click on it a few times, leaving it in the checked state.

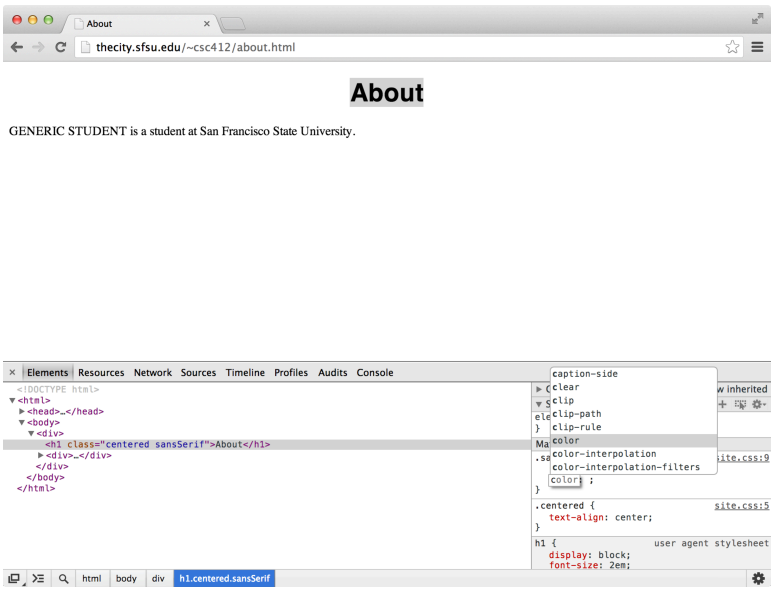

Figure 36. Selector for CSS properties.

8. Click on the close curly brace below the font-family line. A blank line should appear. Type c, and wait a moment. Much like an IDE, a list of CSS properties beginning with the letter 'c' should appear. Use the cursor to select color, then press ↵. Next pressing `tab` should move your cursor to the right of a colon next to color. Type r, and wait for the possible colors available. Select red, and press ↵. Notice what happens to your title.

9. In the Matched CSS portion of the pane, you will see a link in the upper-right corner, indicating something like site.css:9. Click this link. The Sources tab should open in the lower-left pane, showing site.css.

10. In the site.css tab in the sources pane, you can actually see the CSS file that was loaded. Below the color: red; property line, type

```
border-color: black; ↵
border-style: solid; ↵
```

    Observe what happens to your title.

11. The modifications that you have made to the site.css cannot simply be saved back to your server with the current configurations. However, you can save the file to your local file system. Right click in the site.css tab pane, and observe that you can save the file to your local system. You can then upload your site.css file to your local file system if you choose.

12. Press the refresh button. Your page should revert to its original state.

Lab 3 References

Good starting points for debugging, paying special attention to the references at the end:

http://en.wikipedia.org/wiki/Rubber_duck_debugging

http://en.wikipedia.org/wiki/Debugging

Public domain photo of Grace Hopper's bug from

http://upload.wikimedia.org/wikipedia/commons/8/8a/H96566k.jpg

Answer to #14
What you were observing here was a recursive method with no base case. methodB kept calling itself again and again, with a larger and larger value for x, eventually causing the stack to overflow (remember stack frames?). By checking instead if x > 9, x = 10 becomes the base case, and causes the recursion to terminate.