Eric Roberts
CS 106A

Handout #38
February 8, 2010

Debugging

> *As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. We had to discover debugging. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.*
>
> — Maurice Wilkes, 1949

The purpose of this lecture is twofold: to give you a sense of the philosophy of debugging and to teach you how to use some of the practical tools that make debugging easier.

The philosophy of debugging

> *With method and logic one can accomplish anything.*
>
> — Agatha Christie, *Poirot Investigates*, 1924

Debugging is one of the most creative and intellectually challenging aspects of programming. It can also be one of the most frustrating. To a large extent, the problems that people face debugging programs are not so much technical as they are psychological. To turn you into successful debuggers, I have to get you to think in a different way. There is no cookbook approach to debugging, although Nick Parlante's rules in Figure 1 will probably help. What you need is insight, creativity, logic, and determination.

If there is any single aspect of today's lecture that I hope you take with you, it is the idea that the programming process leads you through a series of tasks and roles:

Design	—	Architect
Coding	—	Engineer
Testing	—	Vandal
Debugging	—	Detective

These roles require you to adopt distinct strategies and goals, and it is often difficult to shift your perspective from one to another. Beyond understanding the multiplicity of roles, the main point of today is that, although debugging is extremely difficult, it can be done. It will at times take all of the skill and creativity at your disposal, but you can succeed if you are methodical and don't give up on the task.

I also strongly commend to your attention—not now when you're studying for the midterm and not necessarily even this quarter, but sometime—that you read Robert Pirsig's critically acclaimed novel *Zen and the Art of Motorcycle Maintenance: An Inquiry into Values* (Bantam, 1974), which stands as the best exposition of the art and psychology of debugging ever written. The selections I will read in class today come from Chapter 26.

Using an online debugger

> *There is nothing like first-hand evidence.*
>
> — Sir Arthur Conan Doyle, *A Study in Scarlet*, 1888

Because debugging is a difficult but nonetheless critical task, it is important to learn the tricks of the trade. The most important of these tricks is to get the computer to show you what it's doing, which is the key to debugging. The computer, after all, is there in front

Figure 1. The Eleven Truths of Debugging

1. Intuition and hunches are great—you just have to test them out. When a hunch and a fact collide, the fact wins.

2. Don't look for complex explanations. Even the simplest omission or typo can lead to very weird behavior. Everyone is capable producing extremely simple and obvious errors from time to time. Look at code critically—don't just sweep your eye over that series of simple statements assuming that they are too simple to be wrong.

3. The clue to what is wrong in your code is in the values of your variables and the flow of control. Try to see what the facts are pointing to. The computer is not trying to mislead you. Work from the facts.

4. Be systematic. Be persistent. Don't panic. The bug is not moving around in your code, trying to trick or evade you. It is just sitting in one place, doing the wrong thing in the same way every time.

5. If you code was working a minute ago, but now it doesn't—what was the last thing you changed? This incredibly reliable rule of thumb is the reason your section leader told you to test your code as you go rather than all at once.

6. Do not change your code haphazardly trying to track down a bug. This is sort of like a scientist who changes more than one variable at a time. It makes the observed behavior much more difficult to interpret, and you tend to introduce new bugs.

7. If you find some wrong code which does not seem to be related to the bug you were tracking, fix the wrong code anyway. Many times the wrong code was related to or obscured the bug in a way you had not imagined.

8. You should be able to explain in Sherlock Holmes style the series of facts, tests, and deductions which led you to find a bug. Alternately, if you have a bug but can't pinpoint it, then you should be able to give an argument to a critical third party detailing why each one of your procedures cannot contain the bug. One of these arguments will contain a flaw since one of your procedures does in fact contain a bug. Trying to construct the arguments may help you to see the flaw.

9. Be critical of your beliefs about your code. It's almost impossible to see a bug in a procedure when your instinct is that the procedure is innocent. In that case, only when the facts have proven without question that the procedure is the source of the problem will you be able to see the bug.

10. You need to be systematic, but there is still an enormous amount of room for beliefs, hunches, guesses, etc. Use your intuition about where the bug probably is to direct the order that you check things in your systematic search. Check the procedures you suspect the most first. Good instincts will come with experience.

11. Debugging depends on an objective and reasoned approach. It depends on overall perspective and understanding of the workings of your code. Debugging code is more mentally demanding than writing code. The longer you try to track down a bug without success, the less perspective you tend to have. Realize when you have lost the perspective on your code to debug. Take a break. Get some sleep. You cannot debug when you are not seeing things clearly. Many times a programmer can spend hours late at night hunting for a bug only to finally give up at 4:00A.M. The next day, they find the bug in 10 minutes. What allowed them to find the bug the next day so quickly? Maybe they just needed some sleep and time for perspective. Or maybe their subconscious figured it out while they were asleep. In any case, the "go do something else for a while, come back, and find the bug immediately" scenario happens too often to be an accident.

— Nick Parlante, Stanford University

of you. You can watch it work. You can't ask the computer why it isn't working, but you can have it show you its work as it goes. Modern programming environments like Eclipse come equipped with a **debugger,** which is a special facility for monitoring a program as it runs. By using the Eclipse debugger, for example, you can step through the operation of your program and watch it work. Using the debugger helps you build up a good sense of what your program is doing, and often points the way to the mistake.

To illustrate the operation of the Eclipse debugger in as concrete a way as possible, I will spend the last 15 minutes of today's lecture finding bugs in the `Roulette.java` program, which is reproduced in Figure 2. As the bug icons indicate, the program is buggy. As a programmer, it is your job to figure out why. The remainder of this handout describes the techniques you might use to look for bugs with the help of the Eclipse debugger.

Figure 2. Buggy program intended to play a simplified form of roulette

```
/*
 * File: Roulette.java
 * -------------------
 * This program simulates a small part of the casino game of
 * roulette.
 */

import acm.program.*;
import acm.util.*;

public class Roulette extends ConsoleProgram {

/** Amount of cash with which the player starts */
   private static final int STARTING_MONEY = 100;

/** Amount wagered in each game */
   private static final int WAGER_AMOUNT = 10;

/** Runs the program */
   public void run() {
      giveInstructions();
      playRoulette();
   }

/**
 * Plays roulette until the user runs out of money.
 */
   private void playRoulette() {
      int money = STARTING_MONEY;
      while (money > 0) {
         println("You now have $" + money + ".");
         String bet = readLine("Enter betting category: ");
         int outcome = spinRouletteWheel();
         if (isWinningCategory(outcome, bet)) {
            println("That number is " + bet + " so you win.");
            money += WAGER_AMOUNT;
         } else {
            println("That number is not " + bet + " so you lose.");
            money -= WAGER_AMOUNT;
         }
      }
      println("You ran out of money.");
   }
```