Computer Science 241

Lab 4 (10 points)
Due Sunday, May 6th, 2018 at 10:00 PM

Read all of the instructions. Late work will not be accepted.

Overview

Last week we discussed testing. Once tests have detected a problem, you need to locate and fix the problem. This is known as *debugging*. Every programmer deals with bugs. As one gains experience, it becomes easier to *find them*, *fix them*, and *avoid them in the first place*, but they never entirely go away. Introductory computer science students, however, tend to have an especially difficult time debugging, which can interfere with both learning and the inherent fun of programming. However, the same bug that stalled a student for hours can often be found and fixed within minutes with the help of the instructor. Part of this can be attributed to a fresh pair of eyes, but the primary difference is the instructor's understanding of the art of debugging. This lab is designed to convey some of the fundamental strategies in debugging, with the goal of improving your ability to effectively handle bugs.

There are several kinds of software bugs, but they all have one thing in common: they cause your program to behave in incorrect or unintended ways. Whereas some syntax bugs can be caught by the compiler (all of those compiler errors you love), others cannot be detected automatically. This lab will focus on *logic errors*; that is, when your program dutifully executes the instructions you gave it, but unfortunately your instructions do not produce the intended outcome. No compiler can fully save you from yourself.

So you have a logic bug...

The good news is that most of what you've written is correct (probably). The bad news is that at least one thing you've written isn't correct, and now you have to find it. Inevitably, there will be any number of places where the bug could be.

Stop for a second, and take a deep breath, because this is where many students go horribly wrong. Do not start randomly changing your code, hoping that the next little tweak will solve your problem. Frantic, disorganized debugging is not likely to solve your problem, and will often worsen it. You may also want to add and commit your code in your repository before debugging, so you have a clear record of all debugging changes you made.

Then, carefully and systematically isolate the problem. Whereas when you begin, the problem is "in your program" or "in this class," as you continue to drill down your problem will become "in this particular function" and then, for example, "in this loop." This can be done with increasingly specific tests. As your test smaller and smaller components, you can deem certain parts of your code as more or less likely to be the culprit. Below are three strategies for localizing your problem.

Comment things out

This one is particularly handy for detecting runtime exceptions. If you comment out a particular chunk of code, and the problem still persists, it reduces the odds that your bug is in that chunk. Likewise, if after commenting something out the problem goes away, you're probably on the right track. A related strategy that is to add a `return` statement before the end of a function. Although it won't do everything is supposed to do, the idea is to see if it at least does the first things correctly. You can gradually move the `return` statement further through the function, confirming each time that it works up until that point.

Print statements

What you don't know about your code can certainly hurt you. When there is a bug in your code, often the state of the program (the memory used by the program, including the variables) differs from what you would expect. You can use print statements to keep a closer eye on the state of the program in two key ways:

1. Print the value of variables at key points in your program. When you run your program, watch to see if any of the variables are being set incorrectly. This alone can catch a lot of bugs.

2. Print statements to serve as signposts for where you are in the code (e.g. stating that you've entered in to a function, loop, if statement, etc). This lets you monitor the flow of the program as it runs; watch for any suspicious deviations from the flow you expect.

A debugger tool

A good debugger can be used (often more effectively) to serve the same purposes as print statements: to let you see the state and execution flow of your program. Some debuggers even offer functionality that lets you unit test on the fly, or simulate commenting code out. The next section focuses on one particular debugger for Java: jdb.

Lab Description (debugging with jdb)

The Java Debugger, jdb, is a tool to help you find bugs in your Java programs. Below is a step-by-step walk-through of key jdb functionality; please follow along.

1. Start logging with script using the following command and leave it recording for the remainder of the lab. You will need to turn in the transcript of your commands.:

   ```
   script -a --timing=typescript.timing
   ```

   Note that the `-a` means append, so if you work on this lab in multiple sessions (on the same machine) it will simply append the subsequent session's commands (call this command at the beginning of each session). The `--timing` argument means you will log the timing of your session, which makes it easier for the TA to see the session as you saw it.

2. Compile your code with debugging support:

   ```
   javac -g MergeSort.java
   ```

   The "-g" option tells the compiler to generate extra information useful for debugging.

3. Start the debugger (`jdb`):

   ```
   jdb MergeSort
   ```

   This line starts the debugger with the program symbols (debug information) of your compiled code (`MergeSort.class`).

4. Check out the available options by typing `help` at the prompt `jdb` gives you. This will output a long list of commands that will be very helpful in learning to use jdb. Feel free to type this command anytime in jdb when you cannot remember a command, or want to learn a new command.

5. To run your code inside the debugger, type `run` in the prompt. If your program takes commandline arguments, you can pass them as follows:

   ```
   run <class> <args>
   ```

6. However, you often want to control the pace at which you proceed through running your program, so you can spot where things first go wrong. One way to do this is to step through the program line-by-line. You can do this with the `step` command. When run, it will execute the current line and stop on the next.

7. Sometimes you have a good idea where the bug is, and it might take a prohibitively long time to step line-by-line to get to the area of interest. In these cases, you can you typically want to set *breakpoints*, which tell the debugger where to stop running the code and return control to you. You can set a breakpoint at a particular method with

   ```
   stop in <class>.<method>
   ```

   In that case it will run until this method is called. You can also set a breakpoint at a particular line of code with:

   ```
   stop at <class>:<line>
   ```

   With breakpoints, you don't need to step from the get-go, you can `run` your program and it will return control to you at the first breakpoint.

8. If you want to resume running your program after you have stopped at a breakpoint, you can type `cont` (continue).

9. To list all current breakpoints, type `clear`.

10. To remove a breakpoint, type

    ```
    clear <class>.<method>
    ```

    or

    ```
    clear <class>:<line>
    ```

11. Other assorted useful jdb commands are listed below. Pick at least three of them and try them out in the debugger:

    - You can add a class variable (aka field) to the "watch list" so that you see its value each time it is updated:

      ```
      watch [access|all] <class>.<field_name>
      ```

      Note: You cannot put a "watch" on local variables
    - `next` acts like `step` but it doesn't enter called functions
    - `print <var>` prints the value of a variable
    - `dump <var>` prints object details for an object variable
    - `locals` lists the local variables
    - `where` dumps the stack of the current thread
    - `list` shows you where you are in the code
    - `!!` repeats the last command
    - *Don't forget to check out* `help` *for many other useful commands...*

12. There are (at least) two bugs in `MergeSort.java`. Use the strategies introduced in this lab to find and fix these bugs.

Submission

Once you have finished all of the above, including having found and fixed the bugs, end your script session with `ctrl+d`, then `add`, `commit` and `push` the following files to your course repository `master` branch:

- `lab4/typescript`
- `lab4/typescript.timing`
- `lab4/MergeSort.java` (the version after you have fixed the bugs)

Note that these three files should be within a `lab4` subdirectory of your repository (spelling, spacing and capitalization matter). As always, you can confirm that your code is properly submitted by checking your github repo URL:

`https://github.com/hutchteaching/201820_csci241_username`

Grading

At the deadline a script will automatically clone your repository. Points will be deducted for any problems in your submission, including:

- Missing or incorrectly named files or directories
- Failing to complete all of the walk-through jdb steps.
- Failing to fix the bugs.
- Managing to introducing poor coding style (e.g. inconsistent indentation, silly variable names) into `MergeSort.java`
- Other assorted failure to follow Lab 4 instructions

Acknowledgments

Thanks are owed to Tanzima Islam and Qiang Hao for producing and refining the lab on which lab is modeled.