

ECE498SL: Engineering Software Systems **Spring 2010**
Some Thoughts on Testing and Debugging

This document provides general advice on software development organized as sets of bullet points on particular topics.

1 Getting Started

Few software projects start from scratch. The tips in this section focus on the problem of familiarizing yourself with a complex software system in order to extend it for a new purpose, but may also be useful when you find yourself pitted against a particularly hard bug.

1. Start with a working system.

Don't rush installation of the system on which you plan to build. Look through the options and make sure that you have configured and installed the system appropriately. Execute any regression tests and a few examples to make sure that the installation is working. Trying to extend a broken system is a waste of time because you will have great difficulty in understanding whether you have introduced new bugs or simply have yet to eliminate the ones that were there when you started. If you haven't checked your installation, you must assume that it is broken.

2. Protect your successes.

After overcoming any difficult problem or completing any difficult task, make a copy of all relevant pieces, give them a unique name, and write down the name in your log book along with and explanation of what problems you think you have solved. With source/version control tools, these notes can be entered along with a specific revision of the infrastructure, but physical notes or electronic documents are also often useful. Bug tracking tools can also help you to organize and record your insights, even if you are working alone.

3. Build your understanding as you go.

A solid understanding of the software that you are extending is critical to your long-term success, but can rarely be obtained beforehand or without real experience. Any assumptions that you make about the system but fail to verify may lead to confusion and frustration later. Thus, if you are not sure about how a particular aspect of the system works, try to construct a quick test to help you find out. Adding a little temporary code to isolate the aspect in question is one possibility. Stepping through an operation in a debugger is another method. Allocate a large enough block of time for these activities to ensure that you are able to develop a sound model in your head.

4. Don't worry about making a mess.

If you feel like you can benefit from a few quick experiments, don't worry about whether you've followed the best software engineering rules or naming conventions. Just run a few tests. Be sure that you don't let the mess grow too large, though, nor leave it for a long time. Meaningless names are meaningless names, and trying to decipher them or remember which ones are worth saving later can be time consuming. A temporary new directory can serve both as a way to keep the new files separate and as a way to identify them if you leave them long enough to forget their purpose.

5. You're allowed to roll back to a previous version.

Sometimes, discarding the last round of changes and starting over with a fresh copy of a previous version is easier than trying to figure out where the current version went wrong.

6. Avoid repairing program state by hand.

In theory, you can use a debugger to move backward in program execution time by manually changing the state of the program. The same problem occurs in a different form when working with complex software systems. These systems often depend on large chunks of data. For example, virtual machines depend on their disk images, on any architectural state preserved from earlier execution, and on configuration parameters that control aspects such as device emulation. If these data are somehow corrupted, the guest operating system and applications running under it can fail in bizarre ways. When this happens, the best option is to start again with an earlier version of the state rather than trying to understand the corruption and repair the saved state. Frequent checkpoints of these data help to ensure that you don't lose much work when backtracking is necessary.

7. Use tools to view changes.
Version control tools (as well as `diff`) can help you to identify differences between the current version and a previous version. If you find yourself puzzled about the effect of changes, you might want to verify that the changes are in fact what you think they are. The same tools can be used to view recent changes made by others to related modules, which can help you understand subtle errors that sometimes arise due to incompatibilities between changes spanning several modules.

2 Strategic Debugging

Your time is valuable. A computer's time is not. Keeping these facts in mind as you try to solve your problem can help you avoid frustration and be more effective in your debugging attempts.

1. Choose your debugging tests carefully.
Avoid starting a test without first thinking about the possible outcomes of the test and how you might interpret each of those outcomes. If different outcomes will not help you to identify or to understand a problem, refine the test first.

2. Bugs rarely vanish of their own accord.
Getting lucky with bugs is not common. Unfortunately, getting unlucky by making a bug appear to go away is common. If you don't know why something starts working, you should be worried—it may break again later, after you've built up substantially more confidence in it.

3. Question your assumptions.
If you're stuck, or if you just have some free time to think while an experiment runs, ask yourself what other assumptions you've made implicitly, and write them down. Then question them.

4. Binary search is not just for computers.
If you have identified a problem but are not sure at what point in program execution it occurs, try to check the midpoint of the range, then divide and conquer. For short sequences, the linear checking implied by stepping through code in a debugger is acceptable, but for long sequences, you should leverage your knowledge of algorithms.

5. Sometimes things work exactly once.
The fact that something has worked does not imply that it will work again, nor that it will work if you just change one or two little things. Make a copy before you remove any temporary debugging extras and clean up, then re-run your tests on the clean version. Regression test suites are helpful here, as they allow your computer to do the tests on your behalf. Once the clean version also passes all tests, you can get rid of the messy one.

6. Step back and think.
If you have struggled with a particular problem for a while and find yourself at a loss for the next step, or feel confused by the results of your tests, take a break to help you refocus. Take a walk around the lab, block, campus, or country (depending on the difficulty of the bug). If you're tired, sleep. Even the best tools are no match for human error.

7. Ask for help.
When reading their own work, most people read what they wanted to write rather than what they actually wrote. To avoid this problem, ask a colleague or friend to look over the code with you. Getting an outside perspective on a problem can help you to identify overlooked assumptions and to find mistakes that you obviously couldn't have made. The other person's objectivity is also the reason that code reading and pair programming are effective strategies for avoiding bugs. If you want to preserve or share tentative changes, use preprocessor directives or command-line arguments to turn the changes off by default, then check the code into your version control system.

An Anecdote

In the mid-1990s, I bought a new computer for myself. In those days, almost no one had more than a phone line at home. Internal modems—inside the box housing the computer—were becoming common in new computers, but I already had an external modem, so I decided to use it with my new computer. When I got the computer home and attached the modem, however, I found that the two didn't work together, so I went back to the store.

The problem was cable compatibility between the motherboard and the connector on the side of my new computer. For some reason, the store carried two connector cables that looked identical but had different wiring patterns. Maybe one was meant for a loopback connection—I don't really know. I had the wrong connector in my new computer.

The service person tested my computer with an external modem and saw that it failed to operate, put the other type of connector into the box, tested the modem again and found that it worked, and then took his test connector out, put a new connector in, and sent me off.

When I got home, the modem still didn't work with the computer.

What happened? The new connector was the same as my old connector! He took the working connector out and replaced it with a new instance of the wrong connector.

So I took it back. Another hour wasted. I'd figured out what the service person had done by then, but he hadn't, so he grumbled a bit about my "burning out" wires. Points 3 and 5 apply here: he assumed that since the modem had worked once, the computer had to contain the right connector, despite the fact that he took the working connector out of the box! He ran through the same sequence as before, but I made sure that he didn't make any changes after he had the modem working. After that visit, the computer worked fine for many years.

3 Design

When you have a working infrastructure and are ready to extend it, or are starting a new system from scratch, you may find these tips helpful.

1. Writing code is not your first step.

Before you write code, you need to think about what you're trying to do, how you plan to achieve it, and how your new software will be structured to solve the problem. Use paper, collaboration tools, or anything else that helps to build a sound mental model of how the system operates; without such a model, figuring out what is wrong is much harder, since you don't really know what is right.

2. Build up piece by piece.

Test each piece as you go. Trying to debug many modules simultaneously is a frustrating experience and is usually a pointless waste of your time.

3. Design for clarity and testability; optimize only as necessary.

Making code clear and easily testable is usually much more important than making it fast. Try to avoiding optimizing prematurely: trading human time for speculative reductions in compute time is rarely beneficial, and at the same time often requires more human time for debugging and complex testing.

4. Make use of sanity checks.

Bugs are not limited to complex tasks, nor are invariants always respected by simple code. As you develop your code, you will see opportunities to add simple code to verify that you have achieved your intended goal. Make use of these opportunities: add the code as extra debug code.

5. Interfaces must be testable.

Avoid encapsulating a module in a way that makes future testing difficult or impossible: changes in interface or implementation may break the module, or your original tests may have been inadequate.

6. Develop placeholders and test modules.

Let's say that you want to test a software module M, and that M interfaces with several other modules. Constructing simple versions of the other modules can help you to develop module M more quickly and with fewer bugs. In some cases, these simple versions can then be used as a starting point for developing tests for module M.

7. Spend the time to write tests.

Good tests are necessary to solid development. Sometimes tests can be written before code, but usually the need for more tests becomes clear during the debugging process, and still other tests can be developed to hunt for unknown mistakes.

4 Experimental Analysis

The tips in this section are drawn from experimental scientific principles.

1. Control non-determinism when possible.

Although many sequential programs are entirely deterministic, many others use stochastic methods or randomized algorithms. Non-deterministic behavior can also arise from asynchronous system calls. Determinism is a useful and sometimes necessary feature when trying to understand a bug, as it allows you to re-run the program from the beginning and obtain the same behavior. For programs that include non-determinism through the use of random numbers, saving the random seed used for a given run can make it easier to reproduce bugs, and fixing the seed to a constant value during a debugging session can help to simplify the debugging process and to avoid confusing results. At the same time, use of many seeds can help to expose subtle bugs, and should be used for that purpose after the program seems to be correct.

2. Identify all sources of error.

Before interpreting numerical results, you should spend time thinking about possible sources of error. Some types of error are unavoidable, and systems can rarely be constructed with no error. However, the process of identifying sources of error can help to identify numerical bugs as well as provide a starting point when numerical results seem inconsistent. After identifying sources of error, you should also classify each type as random (with mean 0, implying that averaging can be used to reduce the impact on the results) or systematic (with non-zero mean, implying that some other method must be used to adjust the results if more accuracy is desired). Finally, try to estimate the magnitude of the error (roughly, the variance of the distribution, although more information about the distribution is sometimes useful) so that you have an idea as to the precision with which you should report your results.

An Example

The impact of timer interrupts on program execution time measurements serves as a good example of how the same source of measurement change can be interpreted in several ways, depending on the goal of the experiment and on the system used to make the measurement. Let's say that you are trying to measure the expected execution time of a program running on an otherwise empty system. At first, you use a cycle-accurate architectural simulator and handle system calls (for example, to support I/O) by mapping them to calls on the machine executing the simulator. Any normal operating system takes periodic interrupts from a timer chip in order to virtualize processor resources by scheduling runnable jobs on to real processors. Since the simulator in our scenario does not execute operating system code internally, no timer interrupts occur. One might interpret this lack of timer interrupts as not forming a source of error: after all, you're trying to measure the execution time of your program, not the operating system.

What happens if you decide to simply time the program running on a real system? Your measurement will include timer interrupt handling, and this extra time will not average zero (the contribution can't be negative, and timer interrupts occur every 10 milliseconds in Linux). Are timer interrupts then a source of systematic error? Recognize that no user will execute in a simulator. The execution time that they observe will include timer interrupt handling, and thus perhaps timer interrupts are part of the measurement, not a source of error. In fact, using a simulator in some sense turns the absence of timer interrupts into a negative systematic error for the measure of interest to a user.

Let's think a little harder now. Timer interrupts are asynchronous with respect to program execution. Different executions on the same system may observe different numbers of timer interrupts, and will interleave differently with the interrupt handlers. An average case exists, but measurement of any single execution is unlikely to produce exactly the average. Instead, some small random error will occur around the average because of microarchitectural variations introduced by the interleaving.

What if we decide to execute in a detailed simulator that includes operating system simulation? Now we can capture the timer interrupts and their handling accurately, and even get reproducible timing on interleaving. But we are trying to estimate what a real user sees, so deterministic interleaving is in fact a source of error. It is a single measurement of a random variable, and is thus a source of systematic error with the same magnitude as the variation introduced by the random error in the direct measurement scenario.

As you can see, the type and magnitude of any particular source of error depends on what exactly you are trying to estimate with your measurement and on how you make your measurement.

5 Parallel Execution

Parallel execution introduces many new forms of non-determinism that can make debugging even more difficult. Although some languages attempt to eliminate race conditions (non-determinism that affects program execution), the languages and runtimes used by most parallel code do not. One implication of this fact is that minor changes to a program, such as adding a `printf` or executing under `gdb`, can make a bug vanish.

Acknowledgements

Thanks are due to Junli Gu, John Kelm, and Sanjay Patel for their helpful feedback and suggestions on this document.

Summary of Advice on Testing and Debugging

Getting Started

1. Start with a working system.
2. Protect your successes.
3. Build your understanding as you go.
4. Don't worry about making a mess.
5. You're allowed to roll back to a previous version.
6. Avoid repairing program state by hand.
7. Use tools to view changes.

Design

1. Writing code is not your first step.
2. Build up piece by piece.
3. Design for clarity and testability; optimize only as necessary.
4. Make use of sanity checks.
5. Interfaces must be testable.
6. Develop placeholders and test modules.
7. Spend the time to write tests.

Strategic Debugging

1. Choose your debugging tests carefully.
2. Bugs rarely vanish of their own accord.
3. Question your assumptions.
4. Binary search is not just for computers.
5. Sometimes things work exactly once.
6. Step back and think.
7. Ask for help.

Experimental Analysis

1. Control non-determinism when possible.
2. Identify all sources of error.